目錄

HIT-Alibaba 笔试面试知识整理

本文档使用 Gitbook 制作，Github 仓库地址。

所有引用内容版权归原作者所有。

贡献者：

- skyline75489
- winlandiano
- dodola
- AveryLiu
- JackAlan

HTTP的特性

- HTTP构建于TCP/IP协议之上，默认端口号是80
- HTTP是无连接无状态的

HTTP报文

请求报文

HTTP 协议是以 ASCII 码传输，建立在 TCP/IP 协议之上的应用层规范。规范把 HTTP 请求分为三个部分：状态行、请求头、消息主体。类似于下面这样：

```
<method> <request-URL> <version>
<headers>

<entity-body>
```

HTTP定义了与服务器交互的不同方法，最基本的方法有4种，分别是`GET`，`POST`，`PUT`，`DELETE`。`URL`全称是资源描述符，我们可以这样认为：一个`URL`地址，它用于描述一个网络上的资源，而 HTTP 中的`GET`，`POST`，`PUT`，`DELETE`就对应着对这个资源的查，增，改，删4个操作。

1. GET用于信息获取，而且应该是安全的 和 幂等的。

   所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，GET 请求一般不应产生副作用。就是说，它仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。

   幂等的意味着对同一URL的多个请求应该返回同样的结果。

   GET请求报文示例：

   ```
   GET /books/?sex=man&name=Professional HTTP/1.1
   Host: www.example.com
   User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
   Gecko/20050225 Firefox/1.0.1
   Connection: Keep-Alive
   ```

2. POST表示可能修改变服务器上的资源的请求。

```
POST / HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
Gecko/20050225 Firefox/1.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 40
Connection: Keep-Alive

sex=man&name=Professional
```

3. 注意:

   - GET 可提交的数据量受到URL长度的限制，HTTP 协议规范没有对 URL 长度进行限制。这个限制是特定的浏览器及服务器对它的限制
   - 理论上讲，POST 是没有大小限制的，HTTP 协议规范也没有进行大小限制，出于安全考虑，服务器软件在实现时会做一定限制
   - 参考上面的报文示例，可以发现 GET 和 POST 数据内容是一模一样的，只是位置不同，一个在URL里，一个在 HTTP 包的包体里

POST 提交数据的方式

HTTP 协议中规定 POST 提交的数据必须在 body 部分中，但是协议中没有规定数据使用哪种编码方式或者数据格式。实际上，开发者完全可以自己决定消息主体的格式，只要最后发送的 HTTP 请求满足上面的格式就可以。

但是，数据发送出去，还要服务端解析成功才有意义。一般服务端语言如 php、python 等，以及它们的 framework，都内置了自动解析常见数据格式的功能。服务端通常是根据请求头（headers）中的 Content-Type 字段来获知请求中的消息主体是用何种方式编码，再对主体进行解析。所以说到 POST 提交数据方案，包含了 Content-Type 和消息主体编码方式两部分。下面就正式开始介绍它们：

- `application/x-www-form-urlencoded`

这是最常见的 POST 数据提交方式。浏览器的原生 `<form>` 表单，如果不设置 enctype 属性，那么最终就会以 `application/x-www-form-urlencoded` 方式提交数据。上个小节当中的例子便是使用了这种提交方式。可以看到 body 当中的内容和 GET 请求是完全相同的。

- `multipart/form-data`

这又是一个常见的 POST 数据提交的方式。我们使用表单上传文件时，必须让 `<form>` 表单的 enctype 等于 `multipart/form-data`。直接来看一个请求示例：

```
POST http://www.example.com HTTP/1.1
Content-Type:multipart/form-data; boundary=----WebKitFormBoundaryrGKCBY7qhFd3TrwA

------WebKitFormBoundaryrGKCBY7qhFd3TrwA
Content-Disposition: form-data; name="text"

title
------WebKitFormBoundaryrGKCBY7qhFd3TrwA
Content-Disposition: form-data; name="file"; filename="chrome.png"
Content-Type: image/png

PNG ... content of chrome.png ...
------WebKitFormBoundaryrGKCBY7qhFd3TrwA--
```

这个例子稍微复杂点。首先生成了一个 boundary 用于分割不同的字段，为了避免与正文内容重复，boundary 很长很复杂。然后 `Content-Type` 里指明了数据是以 `multipart/form-data` 来编码，本次请求的 boundary 是什么内容。消息主体里按照字段个数又分为多个结构类似的部分，每部分都是以 --boundary 开始，紧接着是内容描述信息，然后是回车，最后是字段具体内容（文本或二进制）。如果传输的是文件，还要包含文件名和文件类型信息。消息主体最后以 --boundary-- 标示结束。关于 `multipart/form-data` 的详细定义，请前往 RFC1867 查看（或者相对友好一点的 MDN 文档）。

这种方式一般用来上传文件，各大服务端语言对它也有着良好的支持。

上面提到的这两种 POST 数据的方式，都是浏览器原生支持的，而且现阶段标准中原生 `<form>` 表单也只支持这两种方式（通过 `<form>` 元素的 enctype 属性指定，默认为 `application/x-www-form-urlencoded` 。其实 enctype 还支持 text/plain，不过用得非常少）。

随着越来越多的 Web 站点，尤其是 WebApp，全部使用 Ajax 进行数据交互之后，我们完全可以定义新的数据提交方式，例如 `application/json`，`text/xml`，乃至 `application/x-protobuf` 这种二进制格式，只要服务器可以根据 `Content-Type` 和 `Content-Encoding` 正确地解析出请求，都是没有问题的。

响应报文

HTTP 响应与 HTTP 请求相似，HTTP响应也由3个部分构成，分别是：

- 状态行
- 响应头(Response Header)
- 响应正文

状态行由协议版本、数字形式的状态代码、及相应的状态描述，各元素之间以空格分隔。

常见的状态码有如下几种：

- `200 OK` 客户端请求成功
- `301 Moved Permanently` 请求永久重定向
- `302 Moved Temporarily` 请求临时重定向

- `304 Not Modified` 文件未修改，可以直接使用缓存的文件。
- `400 Bad Request` 由于客户端请求有语法错误，不能被服务器所理解。
- `401 Unauthorized` 请求未经授权。这个状态代码必须和WWW-Authenticate报头域一起使用
- `403 Forbidden` 服务器收到请求，但是拒绝提供服务。服务器通常会在响应正文中给出不提供服务的原因
- `404 Not Found` 请求的资源不存在，例如，输入了错误的URL
- `500 Internal Server Error` 服务器发生不可预期的错误，导致无法完成客户端的请求。
- `503 Service Unavailable` 服务器当前不能够处理客户端的请求，在一段时间之后，服务器可能会恢复正常。

下面是一个HTTP响应的例子：

```
HTTP/1.1 200 OK

Server:Apache Tomcat/5.0.12
Date:Mon,6Oct2003 13:23:42 GMT
Content-Length:112

<html>...
```

条件 GET

HTTP 条件 GET 是 HTTP 协议为了减少不必要的带宽浪费，提出的一种方案。详见 RFC2616 。

1. HTTP 条件 GET 使用的时机？

   客户端之前已经访问过某网站，并打算再次访问该网站。

2. HTTP 条件 GET 使用的方法？

   客户端向服务器发送一个包询问是否在上一次访问网站的时间后是否更改了页面，如果服务器没有更新，显然不需要把整个网页传给客户端，客户端只要使用本地缓存即可，如果服务器对照客户端给出的时间已经更新了客户端请求的网页，则发送这个更新了的网页给用户。

   下面是一个具体的发送接受报文示例：

   客户端发送请求：

   ```
   GET / HTTP/1.1
   Host: www.sina.com.cn:80
   If-Modified-Since:Thu, 4 Feb 2010 20:39:13 GMT
   Connection: Close
   ```

第一次请求时，服务器端返回请求数据，之后的请求，服务器根据请求中的 `If-Modified-Since` 字段判断响应文件没有更新，如果没有更新，服务器返回一个 `304 Not Modified` 响应，告诉浏览器请求的资源在浏览器上没有更新，可以使用已缓存的上次获取的文件。

```
HTTP/1.0 304 Not Modified
Date: Thu, 04 Feb 2010 12:38:41 GMT
Content-Type: text/html
Expires: Thu, 04 Feb 2010 12:39:41 GMT
Last-Modified: Thu, 04 Feb 2010 12:29:04 GMT
Age: 28
X-Cache: HIT from sy32-21.sina.com.cn
Connection: close
```

如果服务器端资源已经更新的话，就返回正常的响应。

持久连接

我们知道 HTTP 协议采用“请求-应答”模式，当使用普通模式，即非 Keep-Alive 模式时，每个请求/应答客户和服务器都要新建一个连接，完成之后立即断开连接（HTTP协议为无连接的协议）；当使用 Keep-Alive 模式（又称持久连接、连接重用）时，Keep-Alive 功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive 功能避免了建立或者重新建立连接。

在 HTTP 1.0 版本中，并没有官方的标准来规定 Keep-Alive 如何工作，因此实际上它是被附加到 HTTP 1.0协议上，如果客户端浏览器支持 Keep-Alive ，那么就在HTTP请求头中添加一个字段 Connection: Keep-Alive，当服务器收到附带有 Connection: Keep-Alive 的请求时，它也会在响应头中添加一个同样的字段来使用 Keep-Alive 。这样一来，客户端和服务器之间的HTTP连接就会被保持，不会断开（超过 Keep-Alive 规定的时间，意外断电等情况除外），当客户端发送另外一个请求时，就使用这条已经建立的连接。

在 HTTP 1.1 版本中，默认情况下所有连接都被保持，如果加入 "Connection: close" 才关闭。目前大部分浏览器都使用 HTTP 1.1 协议，也就是说默认都会发起 Keep-Alive 的连接请求了，所以是否能完成一个完整的 Keep-Alive 连接就看服务器设置情况。

由于 HTTP 1.0 没有官方的 Keep-Alive 规范，并且也已经基本被淘汰，以下讨论均是针对 HTTP 1.1 标准中的 Keep-Alive 展开的。

注意：

- HTTP Keep-Alive 简单说就是保持当前的TCP连接，避免了重新建立连接。
- HTTP 长连接不可能一直保持，例如 `Keep-Alive: timeout=5, max=100` ，表示这个TCP通道可以保持5秒，max=100，表示这个长连接最多接收100次请求就断开。

- HTTP是一个无状态协议，这意味着每个请求都是独立的，Keep-Alive没能改变这个结果。另外，Keep-Alive也不能保证客户端和服务器之间的连接一定是活跃的，在HTTP1.1版本中也如此。唯一能保证的就是当连接被关闭时你能得到一个通知，所以不应该让程序依赖于Keep-Alive的保持连接特性，否则会有意想不到的后果。
- 使用长连接之后，客户端、服务端怎么知道本次传输结束呢？两部分：1. 判断传输数据是否达到了Content-Length 指示的大小；2. 动态生成的文件没有 Content-Length ，它是分块传输（chunked），这时候就要根据 chunked 编码来判断，chunked 编码的数据在最后有一个空 chunked 块，表明本次传输数据结束，详见这里。什么是 chunked 分块传输呢？下面我们就来介绍。

Transfer-Encoding

Transfer-Encoding 是一个用来标示 HTTP 报文传输格式的头部值。尽管这个取值理论上可以有很多，但是当前的 HTTP 规范里实际上之定义了一种传输取值——chunked。

如果一个HTTP消息（请求消息或应答消息）的Transfer-Encoding消息头的值为chunked，那么，消息体由数量未定的块组成，并以最后一个大小为0的块为结束。

每一个非空的块都以该块包含数据的字节数（字节数以十六进制表示）开始，跟随一个CRLF（回车及换行），然后是数据本身，最后块CRLF结束。在一些实现中，块大小和CRLF之间填充有白空格（0x20）。

最后一块是单行，由块大小（0），一些可选的填充白空格，以及CRLF。最后一块不再包含任何数据，但是可以发送可选的尾部，包括消息头字段。 消息最后以CRLF结尾。

注意： chunked 和 multipart 两个名词在意义上有类似的地方，不过在 HTTP 协议当中这两个概念则不是一个类别的。multipart 是一种 Content-Type，标示 HTTP 报文内容的类型，而 chunked 是一种传输格式，标示报头将以何种方式进行传输。

HTTP Pipelining（HTTP 管线化）

默认情况下 HTTP 协议中每个传输层连接只能承载一个 HTTP 请求和响应，浏览器会在收到上一个请求的响应之后，再发送下一个请求。在使用持久连接的情况下，某个连接上消息的传递类似于 `请求1 -> 响应1 -> 请求2 -> 响应2 -> 请求3 -> 响应3` 。

HTTP Pipelining（管线化）是将多个 HTTP 请求整批提交的技术，在传送过程中不需等待服务端的回应。使用 HTTP Pipelining 技术之后，某个连接上的消息变成了类似这样 `请求1 -> 请求2 -> 请求3 -> 响应1 -> 响应2 -> 响应3` 。

注意下面几点：

- 管线化机制通过持久连接（persistent connection）完成，仅 HTTP/1.1 支持此技术（HTTP/1.0不支持）

- 只有 GET 和 HEAD 请求可以进行管线化，而 POST 则有所限制
- 初次创建连接时不应启动管线机制，因为对方（服务器）不一定支持 HTTP/1.1 版本的协议
- 管线化不会影响响应到来的顺序，如上面的例子所示，响应返回的顺序并未改变
- HTTP /1.1 要求服务器端支持管线化，但并不要求服务器端也对响应进行管线化处理，只是要求对于管线化的请求不失败即可
- 由于上面提到的服务器端问题，开启管线化很可能并不会带来大幅度的性能提升，而且很多服务器端和代理程序对管线化的支持并不好，因此现代浏览器如 Chrome 和 Firefox 默认并未开启管线化支持

更多关于 HTTP Pipelining 的知识可以参考这里。

会话跟踪

1. 什么是会话？

   客户端打开与服务器的连接发出请求到服务器响应客户端请求的全过程称之为会话。

2. 什么是会话跟踪？

   会话跟踪指的是对同一个用户对服务器的连续的请求和接受响应的监视。

3. 为什么需要会话跟踪？

   浏览器与服务器之间的通信是通过HTTP协议进行通信的，而HTTP协议是"无状态"的协议，它不能保存客户的信息，即一次响应完成之后连接就断开了，下一次的请求需要重新连接，这样就需要判断是否是同一个用户，所以才有会话跟踪技术来实现这种要求。

1. 会话跟踪常用的方法:

   i. URL重写

      URL(统一资源定位符)是Web上特定页面的地址，URL重写的技术就是在URL结尾添加一个附加数据以标识该会话,把会话ID通过URL的信息传递过去，以便在服务器端进行识别不同的用户 。

   ii. 隐藏表单域

      将会话ID添加到HTML表单元素中提交到服务器，此表单元素并不在客户端显示

   iii. Cookie

      Cookie是Web服务器发送给客户端的一小段信息，客户端请求时可以读取该信息发送到服务器端，进而进行用户的识别。对于客户端的每次请求，服务器都会将Cookie发送到客户端,在客户端可以进行保存,以便下次使用。

客户端可以采用两种方式来保存这个Cookie对象，一种方式是保存在客户端内存中，称为临时Cookie，浏览器关闭后这个Cookie对象将消失。另外一种方式是保存在客户机的磁盘上，称为永久Cookie。以后客户端只要访问该网站，就会将这个Cookie再次发送到服务器上，前提是这个Cookie在有效期内，这样就实现了对客户的跟踪。

Cookie是可以被禁止的。

iv. Session:

每一个用户都有一个不同的session，各个用户之间是不能共享的，是每个用户所独享的，在session中可以存放信息。

在服务器端会创建一个session对象，产生一个sessionID来标识这个session对象，然后将这个sessionID放入到Cookie中发送到客户端，下一次访问时，sessionID会发送到服务器，在服务器端进行识别不同的用户。

Session的实现依赖于Cookie，如果Cookie被禁用，那么session也将失效。

跨站攻击

- CSRF（Cross-site request forgery，跨站请求伪造）

CSRF(XSRF) 顾名思义，是伪造请求，冒充用户在站内的正常操作。

例如，一论坛网站的发贴是通过 GET 请求访问，点击发贴之后 JS 把发贴内容拼接成目标 URL 并访问：

```
http://example.com/bbs/create_post.php?title=标题&content=内容
```

那么，我们只需要在论坛中发一帖，包含一链接：

```
http://example.com/bbs/create_post.php?title=我是脑残&content=哈哈
```

只要有用户点击了这个链接，那么他们的帐户就会在不知情的情况下发布了这一帖子。可能这只是个恶作剧，但是既然发贴的请求可以伪造，那么删帖、转帐、改密码、发邮件全都可以伪造。

如何防范 **CSRF** 攻击？可以注意以下几点：

- 关键操作只接受POST请求
- 验证码

CSRF攻击的过程，往往是在用户不知情的情况下构造网络请求。所以如果使用验证码，那么每次操作都需要用户进行互动，从而简单有效的防御了CSRF攻击。

但是如果你在一个网站作出任何举动都要输入验证码会严重影响用户体验，所以验证码一般只出现在特殊操作里面，或者在注册时候使用。

- 检测 Referer

  常见的互联网页面与页面之间是存在联系的，比如你在 `www.baidu.com` 应该是找不到通往 `www.google.com` 的链接的，再比如你在论坛留言，那么不管你留言后重定向到哪里去了，之前的那个网址一定会包含留言的输入框，这个之前的网址就会保留在新页面头文件的 `Referer` 中

  通过检查 `Referer` 的值，我们就可以判断这个请求是合法的还是非法的，但是问题出在服务器不是任何时候都能接受到 `Referer` 的值，所以 Referer Check 一般用于监控 CSRF 攻击的发生，而不用来抵御攻击。

- Token

  目前主流的做法是使用 Token 抵御 CSRF 攻击。下面通过分析 CSRF 攻击来理解为什么 Token 能够有效

  CSRF攻击要成功的条件在于攻击者能够预测所有的参数从而构造出合法的请求。所以根据不可预测性原则，我们可以对参数进行加密从而防止CSRF攻击。

  另一个更通用的做法是保持原有参数不变，另外添加一个参数Token，其值是随机的。这样攻击者因为不知道Token而无法构造出合法的请求进行攻击。

  Token 使用原则

  - Token 要足够随机————只有这样才算不可预测
  - Token 是一次性的，即每次请求成功后要更新Token————这样可以增加攻击难度，增加预测难度
  - Token 要注意保密性————敏感操作使用 post，防止 Token 出现在 URL 中

  注意：过滤用户输入的内容不能阻挡 csrf，我们需要做的是过滤请求的来源。

- XSS（Cross Site Scripting，跨站脚本攻击）

  XSS 全称“跨站脚本”，是注入攻击的一种。其特点是不对服务器端造成任何伤害，而是通过一些正常的站内交互途径，例如发布评论，提交含有 JavaScript 的内容文本。这时服务器端如果没有过滤或转义掉这些脚本，作为内容发布到了页面上，其他用户访问这个页面的时候就会运行这些脚本。

  运行预期之外的脚本带来的后果有很多中，可能只是简单的恶作剧——一个关不掉的窗口：

```
while (true) {
    alert("你关不掉我~");
}
```

也可以是盗号或者其他未授权的操作。

XSS 是实现 CSRF 的诸多途径中的一条，但绝对不是唯一的一条。一般习惯上把通过 XSS 来实现的 CSRF 称为 XSRF。

如何防御 **XSS** 攻击？

理论上，所有可输入的地方没有对输入数据进行处理的话，都会存在XSS漏洞，漏洞的危害取决于攻击代码的威力，攻击代码也不局限于script。防御 XSS 攻击最简单直接的方法，就是过滤用户的输入。

如果不需要用户输入 HTML，可以直接对用户的输入进行 HTML escape 。下面一小段脚本：

```
<script>window.location.href=”http://www.baidu.com”;</script>
```

经过 escape 之后就成了：

```
&lt;script&gt;window.location.href=&quot;http://www.baidu.com&quot;&lt;/script&g
t;
```

它现在会像普通文本一样显示出来，变得无毒无害，不能执行了。

当我们需要用户输入 HTML 的时候，需要对用户输入的内容做更加小心细致的处理。仅仅粗暴地去掉 script 标签是没有用的，任何一个合法 HTML 标签都可以添加 onclick 一类的事件属性来执行 JavaScript。更好的方法可能是，将用户的输入使用 HTML 解析库进行解析，获取其中的数据。然后根据用户原有的标签属性，重新构建 HTML 元素树。构建的过程中，所有的标签、属性都只从白名单中拿取。

参考资料

- 浅谈HTTP中Get与Post的区别
- http请求与http响应详细解析
- HTTP 条件 Get (Conditional Get)
- HTTP中的长连接与短连接
- HTTP Keep-Alive模式
- 分块传输编码
- HTTP 管线化(HTTP pipelining)
- HTTP协议及其POST与GET操作差异 & C#中如何使用POST、GET等
- 四种常见的 POST 提交数据方式

- 会话跟踪
- 总结 XSS 与 CSRF 两种跨站攻击
- CSRF简单介绍与利用方法
- XSS攻击及防御
- 百度百科：HTTP

TCP的特性

- TCP提供一种面向连接的、可靠的字节流服务
- 在一个TCP连接中，仅有两方进行彼此通信。广播和多播不能用于TCP
- TCP使用校验和，确认和重传机制来保证可靠传输
- TCP给数据分节进行排序，并使用累积确认保证数据的顺序不变和非重复
- TCP使用滑动窗口机制来实现流量控制，通过动态改变窗口的大小进行拥塞控制

注意：TCP 并不能保证数据一定会被对方接收到，因为这是不可能的。TCP 能够做到的是，如果有可能，就把数据递送到接收方，否则就（通过放弃重传并且中断连接这一手段）通知用户。因此准确说 TCP 也不是 100% 可靠的协议，它所能提供的是数据的可靠递送或故障的可靠通知。

三次握手与四次挥手

所谓三次握手(Three-way Handshake)，是指建立一个 TCP 连接时，需要客户端和服务器总共发送3个包。

三次握手的目的是连接服务器指定端口，建立 TCP 连接，并同步连接双方的序列号和确认号，交换 TCP 窗口大小信息。在 socket 编程中，客户端执行 `connect()` 时。将触发三次握手。

- 第一次握手(SYN=1, seq=x):

  客户端发送一个 TCP 的 SYN 标志位置1的包，指明客户端打算连接的服务器的端口，以及初始序号 X,保存在包头的序列号(Sequence Number)字段里。

  发送完毕后，客户端进入 `SYN_SEND` 状态。

- 第二次握手(SYN=1, ACK=1, seq=y, ACKnum=x+1):

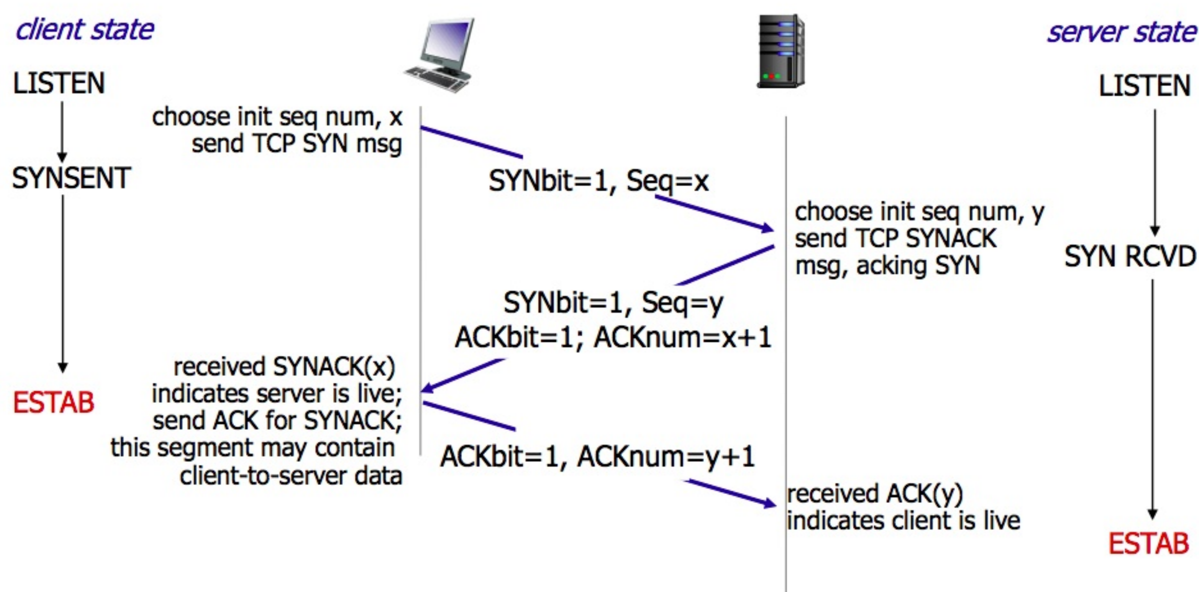
  服务器发回确认包(ACK)应答。即 SYN 标志位和 ACK 标志位均为1。服务器端选择自己 ISN 序列号，放到 Seq 域里，同时将确认序号(Acknowledgement Number)设置为客户的 ISN 加1，即X+1。 发送完毕后，服务器端进入 `SYN_RCVD` 状态。

- 第三次握手(ACK=1，ACKnum=y+1)

  客户端再次发送确认包(ACK)，SYN 标志位为0，ACK 标志位为1，并且把服务器发来 ACK 的序号字段+1，放在确定字段中发送给对方，并且在数据段放写ISN的+1

  发送完毕后，客户端进入 `ESTABLISHED` 状态，当服务器端接收到这个包时，也进入 `ESTABLISHED` 状态，TCP 握手结束。

三次握手的过程的示意图如下：



TCP 的连接的拆除需要发送四个包，因此称为四次挥手(Four-way handshake)，也叫做改进的三次握手。客户端或服务器均可主动发起挥手动作，在 socket 编程中，任何一方执行 `close()` 操作即可产生挥手操作。

- 第一次挥手(FIN=1，seq=x)

  假设客户端想要关闭连接，客户端发送一个 FIN 标志位置为1的包，表示自己已经没有数据可以发送了，但是仍然可以接受数据。

  发送完毕后，客户端进入 `FIN_WAIT_1` 状态。

- 第二次挥手(ACK=1，ACKnum=x+1)

  服务器端确认客户端的 FIN 包，发送一个确认包，表明自己接受到了客户端关闭连接的请求，但还没有准备好关闭连接。

  发送完毕后，服务器端进入 `CLOSE_WAIT` 状态，客户端接收到这个确认包之后，进入 `FIN_WAIT_2` 状态，等待服务器端关闭连接。

- 第三次挥手(FIN=1，seq=y)

  服务器端准备好关闭连接时，向客户端发送结束连接请求，FIN 置为1。

  发送完毕后，服务器端进入 `LAST_ACK` 状态，等待来自客户端的最后一个ACK。

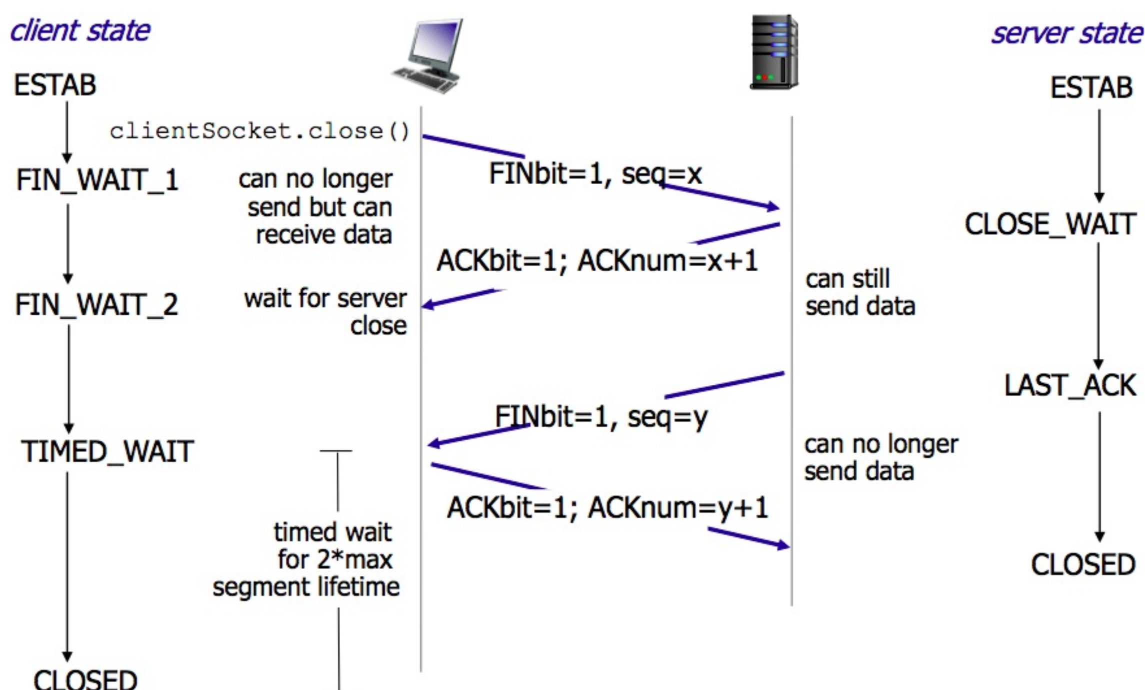
- 第四次挥手(ACK=1，ACKnum=y+1)

  客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 `TIME_WAIT` 状态，等待可能出现的要求重传的 ACK 包。

  服务器端接收到这个确认包之后，关闭连接，进入 `CLOSED` 状态。

客户端等待了某个固定时间（两个最大段生命周期，2MSL，2 Maximum Segment Lifetime）之后，没有收到服务器端的 ACK ，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 `CLOSED` 状态。

四次挥手的示意图如下：



SYN攻击

- 什么是 SYN 攻击（SYN Flood）？

  在三次握手过程中，服务器发送 SYN-ACK 之后，收到客户端的 ACK 之前的 TCP 连接称为半连接(half-open connect)。此时服务器处于 SYN_RCVD 状态。当收到 ACK 后，服务器才能转入 ESTABLISHED 状态.

  SYN 攻击指的是，攻击客户端在短时间内伪造大量不存在的IP地址，向服务器不断地发送SYN包，服务器回复确认包，并等待客户的确认。由于源地址是不存在的，服务器需要不断的重发直至超时，这些伪造的SYN包将长时间占用未连接队列，正常的SYN请求被丢弃，导致目标系统运行缓慢，严重者会引起网络堵塞甚至系统瘫痪。

  SYN 攻击是一种典型的 DoS/DDoS 攻击。

- 如何检测 SYN 攻击？

  检测 SYN 攻击非常的方便，当你在服务器上看到大量的半连接状态时，特别是源IP地址是随机的，基本上可以断定这是一次SYN攻击。在 Linux/Unix 上可以使用系统自带的 netstats 命令来检测 SYN 攻击。

- 如何防御 SYN 攻击？

  SYN攻击不能完全被阻止，除非将TCP协议重新设计。我们所做的是尽可能的减轻SYN攻击的危害，常见的防御 SYN 攻击的方法有如下几种：

  - 缩短超时（SYN Timeout）时间
  - 增加最大半连接数
  - 过滤网关防护
  - SYN cookies技术

参考资料

- 计算机网络：自顶向下方法
- TCP三次握手及四次挥手详细图解
- TCP协议三次握手过程分析
- TCP协议中的三次握手和四次挥手(图解)
- 百度百科：SYN攻击

UDP 简介

UDP 是一个简单的传输层协议。和 TCP 相比，UDP 有下面几个显著特性：

- UDP 缺乏可靠性。UDP 本身不提供确认，序列号，超时重传等机制。UDP 数据报可能在网络中被复制，被重新排序。即 UDP 不保证数据报会到达其最终目的地，也不保证各个数据报的先后顺序，也不保证每个数据报只到达一次
- UDP 数据报是有长度的。每个 UDP 数据报都有长度，如果一个数据报正确地到达目的地，那么该数据报的长度将随数据一起传递给接收方。而 TCP 是一个字节流协议，没有任何（协议上的）记录边界。
- UDP 是无连接的。UDP 客户和服务器之前不必存在长期的关系。UDP 发送数据报之前也不需要经过握手创建连接的过程。
- UDP 支持多播和广播。

IP 协议简介

IP 协议位于 TCP/IP 协议的第三层——网络层。与传输层协议相比，网络层的责任是提供点到点(hop by hop)的服务，而传输层（TCP/UDP）则提供端到端(end to end)的服务。

IP 地址的分类

A类地址

B类地址

C类地址

D 类地址

广播与多播

广播和多播仅用于UDP（TCP是面向连接的）。

- 广播

  一共有四种广播地址：

  1. 受限的广播

     受限的广播地址为255.255.255.255。该地址用于主机配置过程中IP数据报的目的地址，在任何情况下，router不转发目的地址为255.255.255.255的数据报，这样的数据报仅出现在本地网络中。

  2. 指向网络的广播

     指向网络的广播地址是主机号为全1的地址。A类网络广播地址为netid.255.255.255，其中netid为A类网络的网络号。

     一个router必须转发指向网络的广播，但它也必须有一个不进行转发的选择。

  3. 指向子网的广播

     指向子网的广播地址为主机号为全1且有特定子网号的地址。作为子网直接广播地址的IP地址需要了解子网的掩码。例如，router收到128.1.2.255的数据报，当B类网路128.1的子网掩码为255.255.255.0时，该地址就是指向子网的广播地址；但是如果子

网掩码为255.255.254.0，该地址就不是指向子网的广播地址。

4. 指向所有子网的广播

   指向所有子网的广播也需要了解目的网络的子网掩码，以便与指向网络的广播地址区分开来。指向所有子网的广播地址的子网号和主机号为全1.例如，如果子网掩码为255.255.255.0，那么128.1.255.255就是一个指向所有子网的广播地址。

   当前的看法是这种广播是陈旧过时的，更好的方式是使用多播而不是对所有子网的广播。

   广播示例:

   PING 192.168.0.255 (192.168.0.255): 56 data bytes 64 bytes from 192.168.0.107: icmp_seq=0 ttl=64 time=0.199 ms 64 bytes from 192.168.0.106: icmp_seq=0 ttl=64 time=45.357 ms 64 bytes from 192.168.0.107: icmp_seq=1 ttl=64 time=0.203 ms
   64 bytes from 192.168.0.106: icmp_seq=1 ttl=64 time=269.475 ms
   64 bytes from 192.168.0.107: icmp_seq=2 ttl=64 time=0.102 ms 64 bytes from 192.168.0.106: icmp_seq=2 ttl=64 time=189.881 ms

可以看到的确收到了来自两个主机的答复，其中 192.168.0.107 是本机地址。

- 多播

  多播又叫组播，使用D类地址，D类地址分配的28bit均用作多播组号而不再表示其他。

  多播组地址包括1110的最高4bit和多播组号。它们通常可以表示为点分十进制数，范围从224.0.0.0到239.255.255.255。

  多播的出现减少了对应用不感兴趣主机的处理负荷。

  多播的特点：

  - 允许一个或多个发送者（组播源）发送单一的数据包到多个接收者（一次的，同时的）的网络技术
  - 可以大大的节省网络带宽，因为无论有多少个目标地址，在整个网络的任何一条链路上只传送单一的数据包
  - 多播技术的核心就是针对如何节约网络资源的前提下保证服务质量。

```
多播示例：

    PING 224.0.0.1 (224.0.0.1): 56 data bytes
    64 bytes from 192.168.0.107: icmp_seq=0 ttl=64 time=0.081 ms
    64 bytes from 192.168.0.106: icmp_seq=0 ttl=64 time=123.081 ms
    64 bytes from 192.168.0.107: icmp_seq=1 ttl=64 time=0.122 ms
    64 bytes from 192.168.0.106: icmp_seq=1 ttl=64 time=67.312 ms
    64 bytes from 192.168.0.107: icmp_seq=2 ttl=64 time=0.132 ms
    64 bytes from 192.168.0.106: icmp_seq=2 ttl=64 time=447.073 ms
    64 bytes from 192.168.0.107: icmp_seq=3 ttl=64 time=0.132 ms
    64 bytes from 192.168.0.106: icmp_seq=3 ttl=64 time=188.800 ms
```

BGP

- 边界网关协议（BGP）是运行于 TCP 上的一种自治系统的路由协议
- BGP 是唯一一个用来处理像因特网大小的网络的协议，也是唯一能够妥善处理好不相关路由域间的多路连接的协议
- BGP是一种外部网关协议（Exterior Gateway Protocol，EGP），与OSPF、RIP等内部网关协议（Interior Gateway Protocol，IGP）不同，BGP不在于发现和计算路由，而在于控制路由的传播和选择最佳路由
- BGP使用TCP作为其传输层协议（端口号179），提高了协议的可靠性
- BGP既不是纯粹的矢量距离协议，也不是纯粹的链路状态协议
- BGP支持CIDR（Classless Inter-Domain Routing，无类别域间路由）
- 路由更新时，BGP只发送更新的路由，大大减少了BGP传播路由所占用的带宽，适用于在Internet上传播大量的路由信息
- BGP路由通过携带AS路径信息彻底解决路由环路问题
- BGP提供了丰富的路由策略，能够对路由实现灵活的过滤和选择
- BGP易于扩展，能够适应网络新的发展

参考资料

- 多播与广播
- TCP_IP：广播和多播
- 百度百科：BGP

Socket 基本概念

Socket 是对 TCP/IP 协议族的一种封装，是应用层与TCP/IP协议族通信的中间软件抽象层。从设计模式的角度看来，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

Socket 还可以认为是一种网络间不同计算机上的进程通信的一种方法，利用三元组（ip地址，协议，端口）就可以唯一标识网络中的进程，网络中的进程通信可以利用这个标志与其它进程进行交互。

Socket 起源于 Unix ，Unix/Linux 基本哲学之一就是“一切皆文件”，都可以用“打开(open) –> 读写(write/read) –> 关闭(close)”模式来进行操作。因此 Socket 也被处理为一种特殊的文件。

写一个简易的 WebServer

一个简易的 Server 的流程如下：

- 1.建立连接，接受一个客户端连接。
- 2.接受请求，从网络中读取一条 HTTP 请求报文。
- 3.处理请求，访问资源。
- 4.构建响应，创建带有 header 的 HTTP 响应报文。
- 5.发送响应，传给客户端。

省略流程 3，大体的程序与调用的函数逻辑如下：

- socket() 创建套接字
- bind() 分配套接字地址
- listen() 等待连接请求
- accept() 允许连接请求
- read()/write() 数据交换
- close() 关闭连接

代码如下：

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string>
#include <cstring>
#include <iostream>

using namespace std;
```

```
const int port = 9090;
const int buffer_size = 1<<20;
const int method_size = 1<<10;
const int filename_size = 1<<10;
const int common_buffer_size = 1<<10;

void handleError(const string &message);
void requestHandling(int *sock);
void sendError(int *sock);
void sendData(int *sock, char *filename);
void sendHTML(int *sock, char *filename);
void sendJPG(int *sock, char *filename);

int main()
{
    int server_sock;
    int client_sock;

    struct sockaddr_in server_address;
    struct sockaddr_in client_address;

    socklen_t client_address_size;

    server_sock = socket(PF_INET, SOCK_STREAM, 0);

    if (server_sock == -1)
    {
        handleError("socket error");
    }

    memset(&server_address,0,sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(port);

    if(bind(server_sock,(struct sockaddr*)&server_address, sizeof(server_address)) ==
-1){
        handleError("bind error");
    }

    if(listen(server_sock, 5) == -1) {
        handleError("listen error");
    }

    while(true) {
        client_address_size = sizeof(client_address);
        client_sock = accept(server_sock, (struct sockaddr*) &client_address, &client_
address_size);

        if (client_sock == -1) {
            handleError("accept error");
        }
        requestHandling(&client_sock);
    }

    //system("open http://127.0.0.1:9090/index.html");
    close(server_sock);

    return 0;
}

void requestHandling(int *sock){
    int client_sock = *sock;
    char buffer[buffer_size];
    char method[method_size];
    char filename[filename_size];

    read(client_sock, buffer, sizeof(buffer)-1);

    if(!strstr(buffer, "HTTP/")) {
        sendError(sock);
```

```
        close(client_sock);
        return;
    }

    strcpy(method, strtok(buffer," /"));
    strcpy(filename, strtok(NULL, " /"));

    if(0 != strcmp(method, "GET")) {
        sendError(sock);
        close(client_sock);
        return;
    }

    sendData(sock, filename);
}

void sendData(int *sock, char *filename) {
    int client_sock = *sock;
    char buffer[common_buffer_size];
    char type[common_buffer_size];

    strcpy(buffer, filename);

    strtok(buffer, ".");
    strcpy(type, strtok(NULL, "."));

    if(0 == strcmp(type, "html")){
        sendHTML(sock, filename);
    }else if(0 == strcmp(type, "jpg")){
        sendJPG(sock, filename);
    }else{
        sendError(sock);
        close(client_sock);
        return ;
    }
}

void sendHTML(int *sock, char *filename) {
    int client_sock = *sock;
    char buffer[buffer_size];
    FILE *fp;

    char status[] = "HTTP/1.0 200 OK\r\n";
    char header[] = "Server: A Simple Web Server\r\nContent-Type: text/html\r\n\r\n";

    write(client_sock, status, strlen(status));
    write(client_sock, header, strlen(header));

    fp = fopen(filename, "r");
    if(!fp){
        sendError(sock);
        close(client_sock);
        handleError("failed to open file");
        return ;
    }

    fgets(buffer,sizeof(buffer), fp);
    while(!feof(fp)) {
        write(client_sock, buffer, strlen(buffer));
        fgets(buffer, sizeof(buffer), fp);
    }

    fclose(fp);
    close(client_sock);
}

void sendJPG(int *sock, char *filename) {
    int client_sock = *sock;
    char buffer[buffer_size];
    FILE *fp;
    FILE *fw;
```

```
    char status[] = "HTTP/1.0 200 OK\r\n";
    char header[] = "Server: A Simple Web Server\r\nContent-Type: image/jpeg\r\n\r\n";

    write(client_sock, status, strlen(status));
    write(client_sock, header, strlen(header));

    fp = fopen(filename, "rb");
    if(NULL == fp){
        sendError(sock);
        close(client_sock);
        handleError("failed to open file");
        return ;
    }

    fw = fdopen(client_sock, "w");
    fread(buffer, 1, sizeof(buffer), fp);
    while (!feof(fp)){
        fwrite(buffer, 1, sizeof(buffer), fw);
        fread(buffer, 1, sizeof(buffer), fp);
    }

    fclose(fw);
    fclose(fp);
    close(client_sock);
}

void handleError(const string &message) {
    cout<<message;
    exit(1);
}

void sendError(int *sock){
    int client_sock = *sock;

    char status[] = "HTTP/1.0 400 Bad Request\r\n";
    char header[] = "Server: A Simple Web Server\r\nContent-Type: text/html\r\n\r\n";
    char body[] = "<html><head><title>Bad Request</title></head><body><p>400 Bad Reque
st</p></body></html>";

    write(client_sock, status, sizeof(status));
    write(client_sock, header, sizeof(header));
    write(client_sock, body, sizeof(body));
}
```

参考资料

1. Linux Socket编程
2. 揭开 Socket 编程的面纱

单链表

单链表就地翻转

递归算法：

```
void reverse(struct list_node *head)
{
    if(NULL == head || NULL == head->next)
        return;
    reverse1(head->next);
    head->next->next = head;
    head->next = NULL;
}
```

非递归算法：

```
void reverse2(struct list_node *head)
{
    if (NULL == head)
    {
        return;
    }

    list_node *curr = head;
    list_node *next = head->next;
    list_node *prev = NULL;
    while (next != NULL) {
        curr->next = prev;
        prev = curr;
        curr = next;
        next = curr->next;
    }

    curr->next = prev;
}
```

二叉树

二叉树：二叉树是有限个结点的集合，这个集合或者是空集，或者是由一个根结点和两株互不相交的二叉树组成，其中一株叫根的做左子树，另一棵叫做根的右子树。

二叉树的性质：

- 性质1：在二叉树中第 i 层的结点数最多为2^(i-1)（i ≥ 1）
- 性质2：高度为k的二叉树其结点总数最多为2^k－1（ k ≥ 1）
- 性质3：对任意的非空二叉树 T ，如果叶结点的个数为 n0，而其度为 2 的结点数为 n2，则：`n0 = n2 + 1`

满二叉树：深度为k且有2^k －1个结点的二叉树称为满二叉树

完全二叉树：深度为 k 的，有n个结点的二叉树，当且仅当其每个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应，称之为完全二叉树。（除最后一层外，每一层上的节点数均达到最大值；在最后一层上只缺少右边的若干结点）

- 性质4：具有 n 个结点的完全二叉树的深度为 log2n + 1

注意：

- 仅有前序和后序遍历，不能确定一个二叉树，必须有中序遍历的结果

堆

如果一棵完全二叉树的任意一个非终端结点的元素都不小于其左儿子结点和右儿子结点（如果有的话） 的元素，则称此完全二叉树为最大堆。

同样，如果一棵完全二叉树的任意一个非终端结点的元素都不大于其左儿子结点和右儿子结点（如果 有的话）的元素，则称此完全二叉树为最小堆。

最大堆的根结点中的元素在整个堆中是最大的；

最小堆的根结点中的元素在整个堆中是最小的。

哈弗曼树

- 定义：给定n个权值作为n的叶子结点，构造一棵二叉树，若带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman tree)。

- 构造：

假设有n个权值，则构造出的哈夫曼树有n个叶子结点。 n个权值分别设为 w1、w2、…、wn，则哈夫曼树的构造规则为：

1. 将w1、w2、…，wn看成是有 n 棵树的森林(每棵树仅有一个结点)；
2. 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；
3. 从森林中删除选取的两棵树，并将新树加入森林；
4. 重复(2)、(3)步，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。

二叉排序树

二叉排序树（Binary Sort Tree）又称二叉查找树（Binary Search Tree），亦称二叉搜索树。

二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

1. 若左子树不空，则左子树上所有结点的值均小于它的根结点的值；
2. 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值；
3. 左、右子树也分别为二叉排序树；
4. 没有键值相等的节点

二分查找的时间复杂度是O(log(n))，最坏情况下的时间复杂度是O(n)（相当于顺序查找）

平衡二叉树

平衡二叉树（balanced binary tree）,又称 AVL 树。它或者是一棵空树,或者是具有如下性质的二叉树：

1. 它的左子树和右子树都是平衡二叉树，
2. 左子树和右子树的深度之差的绝对值不超过1。

平衡二叉树是对二叉搜索树(又称为二叉排序树)的一种改进。二叉搜索树有一个缺点就是，树的结构是无法预料的，随意性很大，它只与节点的值和插入的顺序有关系，往往得到的是一个不平衡的二叉树。在最坏的情况下，可能得到的是一个单支二叉树，其高度和节点数相同，相当于一个单链表，对其正常的时间复杂度有O(log(n))变成了O(n)，从而丧失了二叉排序树的一些应该有的优点。

B-树

B-树：B-树是一种非二叉的查找树， 除了要满足查找树的特性，还要满足以下结构特性：

一棵 m 阶的B-树：

1. 树的根或者是一片叶子(一个节点的树),或者其儿子数在 2 和 m 之间。

2. 除根外，所有的非叶子结点的孩子数在 m/2 和 m 之间。
3. 所有的叶子结点都在相同的深度。

B-树的平均深度为logm/2(N)。执行查找的平均时间为O(logm)；

Trie 树

Trie 树，又称前缀树，字典树， 是一种有序树，用于保存关联数组，其中的键通常是字符串。与二叉查找树不同，键不是直接保存在节点中，而是由节点在树中的位置决定。一个节点的所有子孙都有相同的前缀，也就是这个节点对应的字符串，而根节点对应空字符串。一般情况下，不是所有的节点都有对应的值，只有叶子节点和部分内部节点所对应的键才有相关的值。

Trie 树查询和插入时间复杂度都是 O(n)，是一种以空间换时间的方法。当节点树较多的时候，Trie 树占用的内存会很大。

Trie 树常用于搜索提示。如当输入一个网址，可以自动搜索出可能的选择。当没有完全匹配的搜索结果，可以返回前缀最相似的可能。

参考资料

- 百度百科：哈弗曼树
- 百度百科：二叉排序树
- 百度百科：平衡二叉树
- 平衡二叉树及其应用场景
- 百度百科：B-树
- 前缀树
- 百度百科：前缀树

哈希表（Hash Table，也叫散列表），是根据关键码值 (Key-Value) 而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。哈希表的实现主要需要解决两个问题，哈希函数和冲突解决。

哈希函数

哈希函数也叫散列函数，它对不同的输出值得到一个固定长度的消息摘要。理想的哈希函数对于不同的输入应该产生不同的结构，同时散列结果应当具有同一性（输出值尽量均匀）和雪崩效应（微小的输入值变化使得输出值发生巨大的变化）。

冲突解决

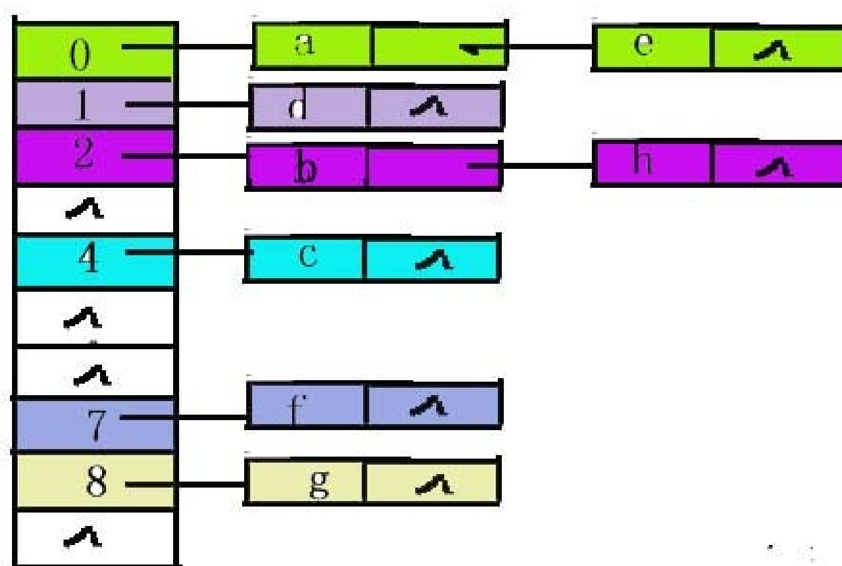
现实中的哈希函数不是完美的，当两个不同的输入值对应一个输出值时，就会产生“碰撞”，这个时候便需要解决冲突。

常见的冲突解决方法有开放定址法，链地址法，建立公共溢出区等。实际的哈希表实现中，使用最多的是链地址法

链地址法

链地址法的基本思想是，为每个 Hash 值建立一个单链表，当发生冲突时，将记录插入到链表中。

例 2 设有 8 个元素 { a,b,c,d,e,f,g,h } ，采用某种哈希函数得到的地址分别为： {0 ， 2 ， 4 ， 1 ， 0 ， 8 ， 7 ， 2} ，当哈希表长度为 10 时，采用链地址法解决冲突的哈希表如下图所示：



参考资料

- 哈希表的 C 实现
- 解决哈希表的冲突-开放地址法和链地址法

排序算法的评价

稳定性

稳定排序算法会依照相等的关键（换言之就是值）维持纪录的相对次序。也就是一个排序算法是稳定的，就是当有两个有相等关键的纪录R和S，且在原本的串行中R出现在S之前，在排序过的串行中R也将会是在S之前。

计算复杂度（最差、平均、和最好表现）

依据串行（list）的大小（n），一般而言，好的表现是O(nlogn)，且坏的行为是O(n2)。对于一个排序理想的表现是O(n)。仅使用一个抽象关键比较运算的排序算法总平均上总是至少需要O(nlogn)。

所有基于比较的排序的时间复杂度至少是 O(nlogn)。

常见排序算法

常见的稳定排序算法有：

- 冒泡排序（Bubble Sort） — $O(n^2)$
- 插入排序（Insertion Sort）— $O(n^2)$
- 桶排序（Bucket Sort）— O(n); 需要 O(k) 额外空间
- 计数排序 (Counting Sort) — O(n+k); 需要 O(n+k) 额外空间
- 合并排序（Merge Sort）— O(nlogn); 需要 O(n) 额外空间
- 二叉排序树排序 （Binary tree sort） — O(n log n) 期望时间; $O(n^2)$最坏时间; 需要 O(n) 额外空间
- 基数排序（Radix sort）— O(n·k); 需要 O(n) 额外空间

常见的不稳定排序算法有：

- 选择排序（Selection Sort）— $O(n^2)$
- 希尔排序（Shell Sort）— O(nlogn)
- 堆排序（Heapsort）— O(nlogn)
- 快速排序（Quicksort）— O(nlogn) 期望时间, $O(n^2)$ 最坏情况; 对于大的、乱数串行一般相信是最快的已知排序

冒泡排序

冒泡排序是最简单最容易理解的排序算法之一，其思想是通过无序区中相邻记录关键字间的比较和位置的交换,使关键字最小的记录如气泡一般逐渐往上“漂浮”直至“水面”。 冒泡排序的复杂度，在最好情况下，即正序有序，则只需要比较n次。故，为O(n) ，最坏情况下，即逆序有序，则需要比较(n-1)+(n-2)+……+1，故，为$O(n^2)$。

乌龟和兔子

在冒泡排序中，最大元素的移动速度是最快的，哪怕一开始最大元素处于序列开头，也可以在一轮内层循环之后，移动到序列末尾。而对于最小元素，每一轮内层循环只能向前挪动一位，如果最小元素在序列末尾，就需要 n-1 次交换才能移动到序列开头。这两种类型的元素分别被称为兔子和乌龟。

代码实现：

```
private static void BubbleSort(int[] array)
{
    for (var i = 0; i < array.Length - 1; i++)  // 若最小元素在序列末尾，需要 n-1 次交换，才能交换到序列开头
    {
        for (var j = 0; j < array.Length - 1; j++)
        {
            if (array[j] > array[j + 1])   // 若这里的条件是 >=，则变成不稳定排序
            {
                Swap(array, j, j+1);
            }
        }
    }
}
```

优化

在非最坏的情况下，冒泡排序过程中，可以检测到整个序列是否已经排序完成，进而可以避免掉后续的循环：

```
private static void BubbleSort(int[] array)
{
    for (var i = 0; i < array.Length - 1; i++)
    {
        var swapped = false;
        for (var j = 0; j < array.Length - 1; j++)
        {
            if (array[j] > array[j + 1])
            {
                Swap(array, j, j+1);
                swapped = true;
            }
        }

        if (!swapped)  // 没有发生交互，证明排序已经完成
        {
            break;
        }
    }
}
```

进一步地，在每轮循环之后，可以确认，最后一次发生交换的位置之后的元素，都是已经排好序的，因此可以不再比较那个位置之后的元素，大幅度减少了比较的次数：

```
private static void BubbleSort(int[] array)
{
    var n = array.Length;
    for (var i = 0; i < array.Length - 1; i++)
    {
        var newn = 0;
        for (var j = 0; j < n - 1; j++)
        {
            if (array[j] > array[j + 1])
            {
                Swap(array, j, j+1);
                newn = j + 1;   // newn 以及之后的元素，都是排好序的
            }
        }

        n = newn;

        if (n == 0)
        {
            break;
        }
    }
}
```

更进一步地，为了优化之前提到的乌龟和兔子问题，可以进行双向的循环，正向循环把最大元素移动到末尾，逆向循环把最小元素移动到最前，这种优化过的冒泡排序，被称为鸡尾酒排序：

```
private static void CocktailSort(int[] array)
{
    var begin = 0;
    var end = array.Length - 1;
    while (begin <= end)
    {
    var newBegin = end;
    var newEnd = begin;

    for (var j = begin; j < end; j++)
    {
        if (array[j] > array[j + 1])
        {
        Swap(array, j, j + 1);
        newEnd = j + 1;
        }
    }

    end = newEnd - 1;

    for (var j = end; j > begin - 1; j--)
    {
        if (array[j] > array[j + 1])
        {
        Swap(array, j, j + 1);
        newBegin = j;
        }
    }

    begin = newBegin + 1;
    }
}
```

插入排序

插入排序也是一个简单的排序算法，它的思想是，每次只处理一个元素，从后往前查找，找到该元素合适的插入位置，最好的情况下，即正序有序(从小到大)，这样只需要比较n次，不需要移动。因此时间复杂度为O(n) ，最坏的情况下，即逆序有序，这样每一个元素就需要比较n次，共有n个元素，因此实际复杂度为$O(n^2)$ 。

算法实现：

```
private static void InsertionSort(int[] array)
{
    int i = 1;
    while (i < array.Length)
    {
    var j = i;
    while (j > 0 && array[j - 1] > array[j])
    {
        Swap(array, j, j - 1);
        j--;
    }

    i++;
    }
}
```

快排

快排是经典的 divide & conquer 问题，如下用于描述快排的思想、伪代码、代码、复杂度计算以及快排的变形。

快排的思想

如下的三步用于描述快排的流程：

- 在数组中随机取一个值作为标兵
- 对标兵左、右的区间进行划分(将比标兵大的数放在标兵的右面，比标兵小的数放在标兵的左面，如果倒序就反过来)
- 重复如上两个过程，直到选取了所有的标兵并划分(此时每个标兵决定的区间中只有一个值，故有序)

伪代码

如下是快排的主体伪代码

```
QUCIKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)
```

如下是用于选取标兵以及划分的伪代码

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] <= x
        i++
        swap A[i] with A[j]
swap A[i+1] with A[j]
return i+1
```

代码

```
func quickSort(inout targetArray: [Int], begin: Int, end: Int) {
    if begin < end {
        let pivot = partition(&targetArray, begin: begin, end: end)
        quickSort(&targetArray, begin: begin, end: pivot - 1)
        quickSort(&targetArray, begin: pivot + 1, end: end)
    }
}

func partition(inout targetArray: [Int], begin: Int, end: Int) -> Int {
    let value = targetArray[end]
    var i = begin - 1
    for j in begin ..< end {
        if  targetArray[j] <= value {
            i += 1;
            swapTwoValue(&targetArray[i], b: &targetArray[j])
        }
    }
    swapTwoValue(&targetArray[i+1], b: &targetArray[end])
    return i+1
}

func swapTwoValue(inout a: Int, inout b: Int) {
    let c = a
    a = b
    b = c
}

var testArray :[Int] = [123,3333,223,231,3121,245,1123]

quickSort(&testArray, begin: 0, end: testArray.count-1)
```

复杂度分析

在最好的情况下，每次 partition 都会把数组一分为二，所以时间复杂度 $T(n) = 2T(n/2) + O(n)$

解为 $T(n) = O(n\log(n))$

在最坏的情况下，数组刚好和想要的结果顺序相同，每次 partition 到的都是当前无序区中最小(或最大)的记录，因此只得到一个比上一次划分少一个记录的子序列。$T(n) = O(n) + T(n-1)$

解为 $T(n) = O(n^2)$

在平均的情况下，快排的时间复杂度是 $O(n\log(n))$

变形

可以利用快排的 PARTITION 思想求数组中第K大元素这样的问题，步骤如下：

- 在数组中随机取一个值作为标兵，左右分化后其顺序为X
- 如果 X == Kth 说明这就是第 K 大的数
- 如果 X > Kth 说明第 K 大的数在标兵左边，继续在左边寻找第 Kth 大的数
- 如果 X < Kth 说明第 K 大的数在标兵右边，继续在右边需找第 Kth - X 大的数

这个问题的时间复杂度是 O(n)

T(n) = n + n/2 + n/4 + ... = O(n)

参考资料

1. 各种基本排序算法的总结
2. 常用排序算法小结
3. 八大排序算法总结
4. QuickSort

洗牌算法

洗牌算法，顾名思义，就是只利用一次循环等概率的取到不同的元素(牌)。

如果元素存在于数组中，即可将每次 random 到的元素 与 最后一个元素进行交换，然后 count--，即可。

这相当于把这个元素删除，代码如下：

```
#include <iostream>
#include <ctime>
using namespace std;

const int maxn = 10;

int a[maxn];

int randomInt(int a) {
    return rand()%a;
}
void swapTwoElement(int*x,int*y) {
     int temp;
     temp=*x;
     *x=*y;
     *y=temp;
}

int main(){
    int count = sizeof(a)/sizeof(int);
    int count_b = count;
    srand((unsigned)time(NULL));
    for (int i = 0; i < count; ++i) { a[i] = i; }
    for (int i = 0; i < count_b; ++i) {
        int random = randomInt(count);
        cout<<a[random]<<" ";
        swapTwoElement(&a[random],&a[count-1]);
        count--;
    }
}
```

贪心算法

建议观看MIT算法导论-贪心算法中的课程。

动态规划

建议观看MIT算法导论-动态规划中的课程。

冯·诺依曼体系结构

1. 计算机处理的数据和指令一律用二进制数表示
2. 顺序执行程序

   计算机运行过程中，把要执行的程序和处理的数据首先存入主存储器（内存），计算机执行程序时，将自动地并按顺序从主存储器中取出指令一条一条地执行，这一概念称作顺序执行程序。

3. 计算机硬件由运算器、控制器、存储器、输入设备和输出设备五大部分组成。

数据的机内表示

二进制表示

1. 机器数

   由于计算机中符号和数字一样,都必须用二进制数串来表示,因此,正负号也必须用0、1来表示。

   用最高位0表示正、1表示负, 这种正负号数字化的机内表示形式就称为“机器数”,而相应的机器外部用正负号表示的数称为“真值”,将一个真值表示成二进制字串的机器数的过程就称为编码。

2. 原码

   原码就是符号位加上真值的绝对值, 即用第一位表示符号, 其余位表示值. 比如如果是8位二进制:

   [+1]原 = 0000 0001

   [-1]原 = 1000 0001

   第一位是符号位. 因为第一位是符号位, 所以8位二进制数的取值范围就是:

   [1111 1111 , 0111 1111]

   即

   [-127 , 127]

   原码是人脑最容易理解和计算的表示方式

3. 反码

   反码的表示方法是:

正数的反码是其本身

负数的反码是在其原码的基础上, 符号位不变，其余各个位取反.

[+1] = [00000001]原 = [00000001]反

[-1] = [10000001]原 = [11111110]反

可见如果一个反码表示的是负数, 人脑无法直观的看出来它的数值. 通常要将其转换成原码再计算

4. 补码

补码的表示方法是:

正数的补码就是其本身

负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1。 (即在反码的基础上+1)

[+1] = [00000001]原 = [00000001]反 = [00000001]补

[-1] = [10000001]原 = [11111110]反 = [11111111]补

对于负数, 补码表示方式也是人脑无法直观看出其数值的. 通常也需要转换成原码在计算其数值.

5. 定点数与浮点数

定点数是小数点固定的数。在计算机中没有专门表示小数点的位，小数点的位置是约定默认的。一般固定在机器数的最低位之后，或是固定在符号位之后。前者称为定点纯整数，后者称为定点纯小数。

定点数表示法简单直观，但是数值表示的范围太小，运算时容易产生溢出。

浮点数是小数点的位置可以变动的数。为增大数值表示范围，防止溢出，采用浮点数表示法。浮点表示法类似于十进制中的科学计数法。

在计算机中通常把浮点数分成阶码和尾数两部分来表示，其中阶码一般用补码定点整数表示，尾数一般用补码或原码定点小数表示。为保证不损失有效数字，对尾数进行规格化处理，也就是平时所说的科学记数法，即保证尾数的最高位为1，实际数值通过阶码进行调整

阶符表示指数的符号位、阶码表示幂次、数符表示尾数的符号位、尾数表示规格化后的小数值。

```
N = 尾数×基数阶码（指数）
```

位(Bit)、字节(Byte)、字(Word)

位："位(bit)"是电子计算机中最小的数据单位。每一位的状态只能是0或1。

字节：8个二进制位构成1个"字节(Byte)"，它是存储空间的基本计量单位。1个字节可以储存1个英文字母或者半个汉字，换句话说，1个汉字占据2个字节的存储空间。

字："字"由若干个字节构成，字的位数叫做字长，不同档次的机器有不同的字长。例如一台8位机，它的1个字就等于1个字节，字长为8位。如果是一台16位机，那么，它的1个字就由2个字节构成，字长为16位。字是计算机进行数据处理和运算的单位。

字节序

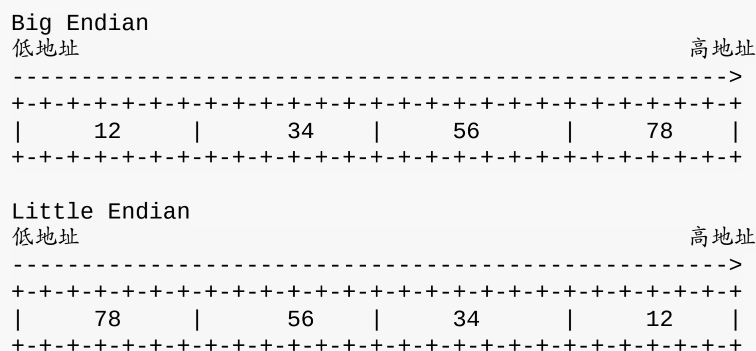
字节顺序是指占内存多于一个字节类型的数据在内存中的存放顺序，通常有小端、大端两种字节顺序。

小端字节序指低字节数据存放在内存低地址处，高字节数据存放在内存高地址处；

大端字节序是高字节数据存放在低地址处，低字节数据存放在高地址处。

基于X86平台的PC机是小端字节序的，而有的嵌入式平台则是大端字节序的。所有网络协议也都是采用big endian的方式来传输数据的。所以有时我们也会把big endian方式称之为网络字节序。

比如数字0x12345678在两种不同字节序CPU中的存储顺序如下所示：



```
Big Endian
低地址                                           高地址
----------------------------------------------------->
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     12      |      34     |      56       |     78    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Little Endian
低地址                                           高地址
----------------------------------------------------->
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     78      |      56     |      34       |     12    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

从上面两图可以看出，采用Big Endian方式存储数据是符合我们人类的思维习惯的。

联合体 `union` 的存放顺序是所有成员都从低地址开始存放，利用该特性，就能判断CPU对内存采用Little-endian还是Big-endian模式读写。

示例代码如下：

```
union test{
     short  i;
     char str[sizeof(short)];
}tt;

void main()
{
     tt.i = 0x0102;
     if(sizeof(short) == 2)
         {
             if(tt.str[0] == 1 && tt.str[1] == 2)
                 printf("大端字节序");
             else if(tt.str[0] = 2 && tt.str[1] == 1)
                 printf("小端字节序");
             else
                 printf("结果未知");
          }
     else
         printf("sizof(short)=%d,不等于2",sizeof(short));
}
```

字节对齐

现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。

- 为什么要进行字节对齐？

  1. 某些平台只能在特定的地址处访问特定类型的数据;
  2. 最根本的原因是效率问题，字节对齐能提高存取数据的速度。

比如有的平台每次都是从偶地址处读取数据,对于一个int型的变量,若从偶地址单元处存放,则只需一个读取周期即可读取该变量，但是若从奇地址单元处存放,则需要2个读取周期读取该变量。

- 字节对齐的原则

  1. 数据成员对齐规则：结构(struct)(或联合(union))的数据成员，第一个数据成员放在offset 为0的地方，以后每个数据成员存储的起始位置要从该成员大小或者成员的子成员大小（只要该成员有子成员，比如说是数组，结构体等）的整数倍开始(比如int在32位机为4字节,则要从4的整数倍地址开始存储。

  2. 结构体作为成员:如果一个结构里有某些结构体成员,则结构体成员要从其内部最大元素大小的整数倍地址开始存储。(struct a里存有struct b,b里有char,int ,double等元素,那b应该从8的整数倍开始存储。)

  3. 收尾工作:结构体的总大小，也就是sizeof的结果，必须是其内部最大成员的整数倍，不足的要补齐。

参考资料

- 百度百科：冯·诺依曼体系结构
- 二进制原码、反码、补码
- 什么是位、字节、字、KB、MB?
- 定点数与浮点数
- 大小字节序
- 浅谈字节序(Byte Order)及其相关操作
- 大小端字节序的判断
- 百度百科：字节对齐
- 五分钟搞定内存对齐

操作系统提供的服务

操作系统的五大功能，分别为：作业管理、文件管理、存储管理、输入输出设备管理、进程及处理机管理

中断与系统调用

中断

所谓的中断就是在计算机执行程序的过程中，由于出现了某些特殊事情，使得CPU暂停对程序的执行，转而去执行处理这一事件的程序。等这些特殊事情处理完之后再回去执行之前的程序。中断一般分为三类：

1. 由计算机硬件异常或故障引起的中断，称为内部异常中断；
2. 由程序中执行了引起中断的指令而造成的中断，称为软中断（这也是和我们将要说明的系统调用相关的中断）；
3. 由外部设备请求引起的中断，称为外部中断。简单来说，对中断的理解就是对一些特殊事情的处理。

与中断紧密相连的一个概念就是中断处理程序了。当中断发生的时候，系统需要去对中断进行处理，对这些中断的处理是由操作系统内核中的特定函数进行的，这些处理中断的特定的函数就是我们所说的中断处理程序了。

另一个与中断紧密相连的概念就是中断的优先级。中断的优先级说明的是当一个中断正在被处理的时候，处理器能接受的中断的级别。中断的优先级也表明了中断需要被处理的紧急程度。每个中断都有一个对应的优先级，当处理器在处理某一中断的时候，只有比这个中断优先级高的中断可以被处理器接受并且被处理。优先级比这个当前正在被处理的中断优先级要低的中断将会被忽略。

典型的中断优先级如下所示：

```
机器错误 > 时钟 > 磁盘 > 网络设备 >  终端 > 软件中断
```

当发生软件中断时，其他所有的中断都可能发生并被处理；但当发生磁盘中断时，就只有时钟中断和机器错误中断能被处理了。

系统调用

在讲系统调用之前，先说下进程的执行在系统上的两个级别：用户级和核心级，也称为用户态和系统态(user mode and kernel mode)。

程序的执行一般是在用户态下执行的，但当程序需要使用操作系统提供的服务时，比如说打开某一设备、创建文件、读写文件等，就需要向操作系统发出调用服务的请求，这就是系统调用。

Linux系统有专门的函数库来提供这些请求操作系统服务的入口，这个函数库中包含了操作系统所提供的对外服务的接口。当进程发出系统调用之后，它所处的运行状态就会由用户态变成核心态。但这个时候，进程本身其实并没有做什么事情，这个时候是由内核在做相应的操作，去完成进程所提出的这些请求。

系统调用和中断的关系就在于，当进程发出系统调用申请的时候，会产生一个软件中断。产生这个软件中断以后，系统会去对这个软中断进行处理，这个时候进程就处于核心态了。

那么用户态和核心态之间的区别是什么呢？（以下区别摘至《UNIX操作系统设计》）

1. 用户态的进程能存取它们自己的指令和数据，但不能存取内核指令和数据（或其他进程的指令和数据）。然而，核心态下的进程能够存取内核和用户地址
2. 某些机器指令是特权指令，在用户态下执行特权指令会引起错误

对此要理解的一个是，在系统中内核并不是作为一个与用户进程平行的估计的进程的集合，内核是为用户进程运行的。

参考资料

- Linux系统的中断、系统调用于调度概述

多任务

在上古时代，CPU 资源十分昂贵，如果让 CPU 只能运行一个程序，那么当 CPU 空闲下来（例如等待 I/O 时），CPU 就空闲下来了。为了让 CPU 得到更好的利用，人们编写了一个监控程序，如果发现某个程序暂时无须使用 CPU 时，监控程序就把另外的正在等待 CPU 资源的程序启动起来，以充分利用 CPU 资源。这种方法被称为 多道程序（**Multiprogramming**）。

对于多道程序来说，最大的问题是程序之间不区分轻重缓急，对于交互式程序来说，对于 CPU 计算时间的需求并不多，但是对于响应速度却有比较高的要求。而对于计算类程序来说则正好相反，对于响应速度要求低，但是需要长时间的 CPU 计算。想象一下我们同时在用浏览器上网和听音乐，我们希望浏览器能够快速响应，同时也希望音乐不停掉。这时候多道程序就没法达到我们的要求了。于是人们改进了多道程序，使得每个程序运行一段时间之后，都主动让出 CPU 资源，这样每个程序在一段时间内都有机会运行一小段时间。这样像浏览器这样的交互式程序就能够快速地被处理，同时计算类程序也不会受到很大影响。这种程序协作方式被称为 分时系统（**Time-Sharing System**）。

在分时系统的帮助下，我们可以边用浏览器边听歌了，但是如果某个程序出现了错误，导致了死循环，不仅仅是这个程序会出错，整个系统都会死机。为了避免这种情况，一个更为先进的操作系统模式被发明出来，也就是我们现在很熟悉的 多任务（**Multi-tasking**）系统。操作系统从最底层接管了所有硬件资源。所有的应用程序在操作系统之上以 进程（**Process**）的方式运行，每个进程都有自己独立的地址空间，相互隔离。CPU 由操作系统统一进行分配。每个进程都有机会得到 CPU，同时在操作系统控制之下，如果一个进程运行超过了一定时间，就会被暂停掉，失去 CPU 资源。这样就避免了一个程序的错误导致整个系统死机。如果操作系统分配给各个进程的运行时间都很短，CPU 可以在多个进程间快速切换，就像很多进程都同时在运行的样子。几乎所有现代操作系统都是采用这样的方式支持多任务，例如 Unix，Linux，Windows 以及 macOS。

进程

进程是一个具有独立功能的程序关于某个数据集合的一次运行活动。它可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体。它不只是程序的代码，还包括当前的活动，通过程序计数器的值和处理寄存器的内容来表示。

进程的概念主要有两点：第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时，它才能成为一个活动的实体，我们称其为进程。

进程的基本状态

1. 等待态：等待某个事件的完成；
2. 就绪态：等待系统分配处理器以便运行；
3. 运行态：占有处理器正在运行。

运行态→等待态 往往是由于等待外设，等待主存等资源分配或等待人工干预而引起的。

等待态→就绪态 则是等待的条件已满足，只需分配到处理器后就能运行。

运行态→就绪态 不是由于自身原因，而是由外界原因使运行状态的进程让出处理器，这时候就变成就绪态。例如时间片用完，或有更高优先级的进程来抢占处理器等。

就绪态→运行态 系统按某种策略选中就绪队列中的一个进程占用处理器，此时就变成了运行态

进程调度

调度种类

高级、中级和低级调度作业从提交开始直到完成，往往要经历下述三级调度：

- 高级调度：(High-Level Scheduling)又称为作业调度，它决定把后备作业调入内存运行；
- 中级调度：(Intermediate-Level Scheduling)又称为在虚拟存储器中引入，在内、外存对换区进行进程对换。
- 低级调度：(Low-Level Scheduling)又称为进程调度，它决定把就绪队列的某进程获得CPU；

非抢占式调度与抢占式调度

- 非抢占式

  分派程序一旦把处理机分配给某进程后便让它一直运行下去，直到进程完成或发生进程调度进程调度某事件而阻塞时，才把处理机分配给另一个进程。

- 抢占式

  操作系统将正在运行的进程强行暂停，由调度程序将CPU分配给其他就绪进程的调度方式。

调度策略的设计

响应时间: 从用户输入到产生反应的时间

周转时间: 从任务开始到任务结束的时间

CPU任务可以分为交互式任务和批处理任务，调度最终的目标是合理的使用CPU，使得交互式任务的响应时间尽可能短，用户不至于感到延迟，同时使得批处理任务的周转时间尽可能短，减少用户等待的时间。

调度算法

1. FIFO或First Come, First Served (FCFS)

   调度的顺序就是任务到达就绪队列的顺序。

   公平、简单(FIFO队列)、非抢占、不适合交互式。未考虑任务特性，平均等待时间可以缩短

2. Shortest Job First (SJF)

   最短的作业(CPU区间长度最小)最先调度。

   可以证明，SJF可以保证最小的平均等待时间。

   - Shortest Remaining Job First (SRJF)

     SJF的可抢占版本，比SJF更有优势。

   SJF(SRJF): 如何知道下一CPU区间大小？根据历史进行预测: 指数平均法。

1. 优先权调度

   每个任务关联一个优先权，调度优先权最高的任务。

   注意：优先权太低的任务一直就绪，得不到运行，出现“饥饿”现象。

   FCFS是RR的特例，SJF是优先权调度的特例。这些调度算法都不适合于交互式系统。

2. Round-Robin(RR)

   设置一个时间片，按时间片来轮转调度（“轮叫”算法）

   优点: 定时有响应，等待时间较短；缺点: 上下文切换次数较多；

   如何确定时间片？

   时间片太大，响应时间太长；吞吐量变小，周转时间变长；当时间片过长时，退化为FCFS。

3. 多级队列调度

   - 按照一定的规则建立多个进程队列
   - 不同的队列有固定的优先级（高优先级有抢占权）
   - 不同的队列可以给不同的时间片和采用不同的调度方法

   存在问题1：没法区分I/O bound和CPU bound；

存在问题2：也存在一定程度的“饥饿”现象；

4. 多级反馈队列

   在多级队列的基础上，任务可以在队列之间移动，更细致的区分任务。

   可以根据“享用”CPU时间多少来移动队列，阻止“饥饿”。

   最通用的调度算法，多数OS都使用该方法或其变形，如UNIX、Windows等。

进程同步

临界资源与临界区

在操作系统中，进程是占有资源的最小单位（线程可以访问其所在进程内的所有资源，但线程本身并不占有资源或仅仅占有一点必须资源）。但对于某些资源来说，其在同一时间只能被一个进程所占用。这些一次只能被一个进程所占用的资源就是所谓的临界资源。典型的临界资源比如物理上的打印机，或是存在硬盘或内存中被多个进程所共享的一些变量和数据等(如果这类资源不被看成临界资源加以保护，那么很有可能造成丢数据的问题)。

对于临界资源的访问，必须是互斥进行。也就是当临界资源被占用时，另一个申请临界资源的进程会被阻塞，直到其所申请的临界资源被释放。而进程内访问临界资源的代码被成为临界区。

对于临界区的访问过程分为四个部分：

1. 进入区:查看临界区是否可访问，如果可以访问，则转到步骤二，否则进程会被阻塞
2. 临界区:在临界区做操作
3. 退出区:清除临界区被占用的标志
4. 剩余区：进程与临界区不相关部分的代码

解决临界区问题可能的方法：

1. 一般软件方法
2. 关中断方法
3. 硬件原子指令方法
4. 信号量方法

信号量

信号量是一个确定的二元组（s，q），其中s是一个具有非负初值的整形变量，q是一个初始状态为空的队列，整形变量s表示系统中某类资源的数目：

- 当其值 ≥ 0 时，表示系统中当前可用资源的数目
- 当其值 ＜ 0 时，其绝对值表示系统中因请求该类资源而被阻塞的进程数目

除信号量的初值外，信号量的值仅能由P操作和V操作更改，操作系统利用它的状态对进程和资源进行管理

P操作：

P操作记为P(s)，其中s为一信号量，它执行时主要完成以下动作：

```
s.value = s.value - 1；  /*可理解为占用1个资源，若原来就没有则记帐“欠”1个*/
```

若s.value ≥ 0，则进程继续执行，否则（即s.value < 0），则进程被阻塞，并将该进程插入到信号量s的等待队列s.queue中

说明：实际上，P操作可以理解为分配资源的计数器，或是使进程处于等待状态的控制指令

V操作：

V操作记为V(s)，其中s为一信号量，它执行时，主要完成以下动作：

```
s.value = s.value + 1；/*可理解为归还1个资源，若原来就没有则意义是用此资源还1个欠帐*/
```

若s.value > 0，则进程继续执行，否则（即s.value ≤ 0）,则从信号量s的等待队s.queue中移出第一个进程，使其变为就绪状态，然后返回原进程继续执行

说明：实际上，V操作可以理解为归还资源的计数器，或是唤醒进程使其处于就绪状态的控制指令

信号量方法实现：生产者 − 消费者互斥与同步控制

```
semaphore fullBuffers = 0; /*仓库中已填满的货架个数*/
semaphore emptyBuffers = BUFFER_SIZE;/*仓库货架空闲个数*/
semaphore mutex = 1; /*生产-消费互斥信号*/

Producer()
{
    while(True)
    {
       /*生产产品item*/
       emptyBuffers.P();
       mutex.P();
       /*item存入仓库buffer*/
       mutex.V();
       fullBuffers.V();
    }
}

Consumer()
{
    while(True)
    {
        fullBuffers.P();
        mutex.P();
        /*从仓库buffer中取产品item*/
        mutex.V();
        emptyBuffers.V();
        /*消费产品item*/
    }
}
```

使用pthread实现的生产者—消费者模型：

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>
#define BUFFER_SIZE 10

static int buffer[BUFFER_SIZE] = { 0 };
static int count = 0;

pthread_t consumer, producer;
pthread_cond_t cond_producer, cond_consumer;
pthread_mutex_t mutex;

void* consume(void* _){
  while(1){
    pthread_mutex_lock(&mutex);
    while(count == 0){
      printf("empty buffer, wait producer\n");
      pthread_cond_wait(&cond_consumer, &mutex);
    }

    count--;
    printf("consume a item\n");
    pthread_mutex_unlock(&mutex);
    pthread_cond_signal(&cond_producer);
    //pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}

void* produce(void* _){
  while(1){
    pthread_mutex_lock(&mutex);
    while(count == BUFFER_SIZE){
      printf("full buffer, wait consumer\n");
      pthread_cond_wait(&cond_producer, &mutex);
```

```
    }

    count++;
    printf("produce a item.\n");
    pthread_mutex_unlock(&mutex);
    pthread_cond_signal(&cond_consumer);
    //pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}

int main() {

  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&cond_consumer, NULL);
  pthread_cond_init(&cond_producer, NULL);

  int err = pthread_create(&consumer, NULL, consume, (void*)NULL);
  if(err != 0){
    printf("consumer thread created failed\n");
    exit(1);
  }

  err = pthread_create(&producer, NULL, produce, (void*)NULL);
  if(err != 0){
    printf("producer thread created failed\n");
    exit(1);
  }

  pthread_join(producer, NULL);
  pthread_join(consumer, NULL);

  //sleep(1000);

  pthread_cond_destroy(&cond_consumer);
  pthread_cond_destroy(&cond_producer);
  pthread_mutex_destroy(&mutex);


  return 0;
}
```

死锁

死锁: 多个进程因循环等待资源而造成无法执行的现象。

死锁会造成进程无法执行，同时会造成系统资源的极大浪费(资源无法释放)。

死锁产生的4个必要条件：

- 互斥使用(Mutual exclusion)

  指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

- 不可抢占(No preemption)

  指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

- 请求和保持(Hold and wait)

指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

- 循环等待(Circular wait) 指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

死锁避免——银行家算法

思想: 判断此次请求是否造成死锁若会造成死锁，则拒绝该请求

进程间通信

本地进程间通信的方式有很多，可以总结为下面四类：

- 消息传递（管道、FIFO、消息队列）
- 同步（互斥量、条件变量、读写锁、文件和写记录锁、信号量）
- 共享内存（匿名的和具名的）
- 远程过程调用（Solaris门和Sun RPC）

线程

多进程解决了前面提到的多任务问题。然而很多时候不同的程序需要共享同样的资源（文件，信号量等），如果全都使用进程的话会导致切换的成本很高，造成 CPU 资源的浪费。于是就出现了线程的概念。

线程，有时被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。一个标准的线程由线程ID，当前指令指针(PC），寄存器集合和堆栈组成。

线程具有以下属性：

1. 轻型实体 线程中的实体基本上不拥有系统资源，只是有一点必不可少的、能保证独立运行的资源。线程的实体包括程序、数据和TCB。线程是动态概念，它的动态特性由线程控制块TCB（Thread Control Block）描述。

2. 独立调度和分派的基本单位。

   在多线程OS中，线程是能独立运行的基本单位，因而也是独立调度和分派的基本单位。由于线程很“轻”，故线程的切换非常迅速且开销小（在同一进程中的）。

3. 可并发执行。 在一个进程中的多个线程之间，可以并发执行，甚至允许在一个进程中所有线程都能并发执行；同样，不同进程中的线程也能并发执行，充分利用和发挥了处理机与外围设备并行工作的能力。

4. 共享进程资源。 在同一进程中的各个线程，都可以共享该进程所拥有的资源，这首先表现在：所有线程都具有相同的地址空间（进程的地址空间），这意味着，线程可以访问该地址空间的每一个虚地址；此外，还可以访问进程所拥有的已打开文件、定时器、信号量等。由于同一个进程内的线程共享内存和文件，所以线程之间互相通信不必调用内核。 线程共享的环境包括：进程代码段、进程的公有数据(利用这些共享的数据，线程很容易的实现相互之间的通讯)、进程打开的文件描述符、信号的处理器、进程的当前目录和进程用户ID与进程组ID。

协程

协程，又称微线程，纤程。英文名Coroutine。

协程可以理解为用户级线程，协程和线程的区别是：线程是抢占式的调度，而协程是协同式的调度，协程避免了无意义的调度，由此可以提高性能，但也因此，程序员必须自己承担调度的责任，同时，协程也失去了标准线程使用多CPU的能力。

使用协程(python)改写生产者-消费者问题：

```
import time

def consumer():
    r = ''
    while True:
        n = yield r
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        time.sleep(1)
        r = '200 OK'

def produce(c):
    next(c)                   #python 3.x中使用next(c)，python 2.x中使用c.next()
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()

if __name__=='__main__':
    c = consumer()
    produce(c)
```

可以看到，使用协程不再需要显式地对锁进行操作。

IO多路复用

基本概念

IO多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。IO多路复用适用如下场合：

1. 当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用I/O复用。
2. 当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。
3. 如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用。
4. 如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用。
5. 如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用。

与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

常见的IO复用实现

- select(Linux/Windows/BSD)
- epoll(Linux)
- kqueue(BSD/Mac OS X)

参考资料

- 浅谈进程同步与互斥的概念
- 进程间同步——信号量
- 百度百科：线程
- 进程、线程和协程的理解
- 协程
- IO多路复用之select总结
- 银行家算法

内存管理基础

程序可执行文件的结构

一个程序的可执行文件在内存中的结果，从大的角度可以分为两个部分：只读部分和可读写部分。只读部分包括程序代码（.text）和程序中的常量（.rodata）。可读写部分（也就是变量）大致可以分成下面几个部分：

- `.data` ：初始化了的全局变量和静态变量
- `.bss` ：即 Block Started by Symbol， 未初始化的全局变量和静态变量（这个我感觉上课真的没讲过啊我去。。。）
- `heap` ：堆，使用 malloc, realloc, 和 free 函数控制的变量，堆在所有的线程，共享库，和动态加载的模块中被共享使用
- `stack` ：栈，函数调用时使用栈来保存函数现场，自动变量（即生命周期限制在某个 scope 的变量）也存放在栈中。

下面就各个分区具体解释一下：

data 和 bss 区

这两个区经常放在一起说，因为他们都是用来存储全局变量和静态变量的，区别在于 data 区存放的是初始化过的， bss 区存放的是没有初始化过的，例如：

```
int val = 3;
char string[] = "Hello World";
```

这两个变量的值会一开始被存储在 .text 中（因为值是写在代码里面的），在程序启动时会拷贝到 .data 去区中。

而不初始化的话，像下面这样：

```
static int i;
```

这个变量就会被放在 bss 区中。

答疑一 静态变量和全局变量

这两个概念都是很常见的概念，又经常在一起使用，很容易造成混淆。

`全局变量` ：在一个代码文件（具体说应该一个 translation unit/compilation unit)）当中，一个变量要么定义在函数中，要么定义在在函数外面。当定义在函数外面时，这个变量就有了全局作用域，成为了全局变量。全局变量不光意味着这个变量可以在整个文件中使用，也意味着这个变量可以在其他文件中使用（这种叫做 external linkage）。当有如下两个文件时；

a.c

```
#include <stdio.h>

int a;

int compute(void);

int main()
{
    a = 1;
    printf("%d %d\n", a, compute());
    return 0;
}
```

b.c

```
int a;

int compute(void)
{
    a = 0;
    return a;
}
```

在 Link 过程中会产生重复定义错误，因为有两个全局的 `a` 变量，Linker 不知道应该使用哪一个。为了避免这种情况，就需要引入 static。

`静态变量`： 指使用 static 关键字修饰的变量，static 关键字对变量的作用域进行了限制，具体的限制如下：

- 在函数外定义：全局变量，但是只在当前文件中可见（叫做 internal linkage）
- 在函数内定义：全局变量，但是只在此函数内可见（同时，在多次函数调用中，变量的值不会丢失）
- （C++）在类中定义：全局变量，但是只在此类中可见

对于全局变量来说，为了避免上面提到的重复定义错误，我们可以在一个文件中使用 static，另一个不使用。这样使用 static 的就会使用自己的 `a` 变量，而没有用 static 的会使用全局的 `a` 变量。当然，最好两个都使用 static，避免更多可能的命名冲突。

注意：'静态'这个中文翻译实在是有些莫名其妙，给人的感觉像是不可改变的，而实际上 static 跟不可改变没有关系，不可改变的变量使用 const 关键字修饰，注意不要混淆。

Bonus 部分 —— *extern*： extern 是 C 语言中另一个关键字，用来指示变量或函数的定义在别的文件中，使用 extern 可以在多个源文件中共享某个变量，例如这里的例子。 extern 跟 static 在含义上是“水火不容”的，一个表示不能在别的地方用，一个表示要去别的地方找。如果同时使用的话，有两种情况，一种是先使用 static，后使用 extern ，即：

```
static int m;
extern int m;
```

这种情况，后面的 m 实际上就是前面的 m 。如果反过来：

```
extern int m;
static int m;
```

这种情况的行为是未定义的，编译器也会给出警告。

答疑二 程序在内存和硬盘上不同的存在形式

这里我们提到的几个区，是指程序在内存中的存在形式。和程序在硬盘上存储的格式不是完全对应的。程序在硬盘上存储的格式更加复杂，而且是和操作系统有关的，具体可以参考这里。一个比较明显的例子可以帮你区分这个差别：之前我们提到过未定义的全局变量存储在 `.bss` 区，这个区域不会占用可执行文件的空间（一般只存储这个区域的长度），但是却会占用内存空间。这些变量没有定义，因此可执行文件中不需要存储（也不知道）它们的值，在程序启动过程中，它们的值会被初始化成 0 ，存储在内存中。

栈

栈是用于存放本地变量，内部临时变量以及有关上下文的内存区域。程序在调用函数时，操作系统会自动通过压栈和弹栈完成保存函数现场等操作，不需要程序员手动干预。

栈是一块连续的内存区域，栈顶的地址和栈的最大容量是系统预先规定好的。能从栈获得的空间较小。如果申请的空间超过栈的剩余空间时，例如递归深度过深，将提示stackoverflow。

栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。

堆

堆是用于存放除了栈里的东西之外所有其他东西的内存区域，当使用 `malloc` 和 `free` 时就是在操作堆中的内存。对于堆来说，释放工作由程序员控制，容易产生memory leak。

堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，永远都不可能有一个内存块从栈中间弹出。

堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

计算机底层并没有对堆的支持，堆则是C/C++函数库提供的，同时由于上面提到的碎片问题，都会导致堆的效率比栈要低。

内存分配

- 虚拟地址：用户编程时将代码（或数据）分成若干个段，每条代码或每个数据的地址由段名称 + 段内相对地址构成，这样的程序地址称为虚拟地址
- 逻辑地址：虚拟地址中，段内相对地址部分称为逻辑地址
- 物理地址：实际物理内存中所看到的存储地址称为物理地址

- 逻辑地址空间：在实际应用中，将虚拟地址和逻辑地址经常不加区分，通称为逻辑地址。逻辑地址的集合称为逻辑地址空间

- 线性地址空间：CPU地址总线可以访问的所有地址集合称为线性地址空间
- 物理地址空间：实际存在的可访问的物理内存地址集合称为物理地址空间

- MMU(Memery Management Unit内存管理单元)：实现将用户程序的虚拟地址（逻辑地址）→ 物理地址映射的CPU中的硬件电路

- 基地址：在进行地址映射时，经常以段或页为单位并以其最小地址（即起始地址）为基值来进行计算
- 偏移量：在以段或页为单位进行地址映射时，相对于基地址的地址值

虚拟地址先经过分段机制映射到线性地址，然后线性地址通过分页机制映射到物理地址。

虚拟内存

- 请求调页，也称按需调页，即对不在内存中的“页”，当进程执行时要用时才调入，否则有可能到程序结束时也不会调入

页面置换算法

- FIFO算法

  先入先出，即淘汰最早调入的页面。

- OPT(MIN)算法

  选未来最远将使用的页淘汰，是一种最优的方案，可以证明缺页数最小。

  可惜，MIN需要知道将来发生的事，只能在理论中存在，实际不可应用。

- LRU(Least-Recently-Used)算法

  用过去的历史预测将来，选最近最长时间没有使用的页淘汰(也称最近最少使用)。

LRU准确实现：计数器法，页码栈法。

由于代价较高，通常不使用准确实现，而是采用近似实现，例如Clock算法。

内存抖动现象：页面的频繁更换，导致整个系统效率急剧下降，这个现象称为内存抖动（或颠簸）。抖动一般是内存分配算法不好，内存太小引或者程序的算法不佳引起的。

Belady现象：对有的页面置换算法，页错误率可能会随着分配帧数增加而增加。

FIFO会产生Belady异常。

栈式算法无Belady异常，LRU，LFU（最不经常使用），OPT都属于栈式算法。

参考资料

- https://en.wikipedia.org/wiki/Data_segment
- https://stackoverflow.com/questions/12798486/bss-segment-in-c/12799389#12799389
- https://stackoverflow.com/questions/7837190/c-c-global-vs-static-global
- https://stackoverflow.com/questions/572547/what-does-static-mean-in-a-c-program/572550#572550

磁盘调度

磁盘访问延迟 = 队列时间 + 控制器时间 + 寻道时间 + 旋转时间 + 传输时间

磁盘调度的目的是减小延迟，其中前两项可以忽略，寻道时间是主要矛盾。

磁盘调度算法

- FCFS

  先进先出的调度策略，这个策略具有公平的优点，因为每个请求都会得到处理，并且是按照接收到的顺序进行处理。

- SSTF(Shortest-seek-time First 最短寻道时间优先)

  选择使磁头从当前位置开始移动最少的磁盘I/O请求，所以 SSTF 总是选择导致最小寻道时间的请求。

  总是选择最小寻找时间并不能保证平均寻找时间最小，但是能提供比 FCFS 算法更好的性能，会存在饥饿现象。

- SCAN

  SSTF+中途不回折，每个请求都有处理机会。

  SCAN 要求磁头仅仅沿一个方向移动，并在途中满足所有未完成的请求，直到它到达这个方向上的最后一个磁道，或者在这个方向上没有其他请求为止。

  由于磁头移动规律与电梯运行相似，SCAN 也被称为电梯算法。

  SCAN 算法对最近扫描过的区域不公平，因此，它在访问局部性方面不如 FCFS 算法和 SSTF 算法好。

- C-SCAN

  SCAN+直接移到另一端，两端请求都能很快处理。

  把扫描限定在一个方向，当访问到某个方向的最后一个磁道时，磁道返回磁盘相反方向磁道的末端，并再次开始扫描。

  其中“C”是Circular（环）的意思。

- LOOK 和 C-LOOK

  采用SCAN算法和C-SCAN算法时磁头总是严格地遵循从盘面的一端到另一端，显然，在实际使用时还可以改进，即磁头移动只需要到达最远端的一个请求即可返回，不需要到达磁盘端点。这种形式的SCAN算法和C-SCAN算法称为LOOK和C-LOOK调度。这是因

为它们在朝一个给定方向移动前会查看是否有请求。

文件系统

分区表

- MBR：支持最大卷为2 TB（Terabytes）并且每个磁盘最多有4个主分区（或3个主分区，1个扩展分区和无限制的逻辑驱动器）
- GPT：支持最大卷为18EB（Exabytes）并且每磁盘的分区数没有上限，只受到操作系统限制（由于分区表本身需要占用一定空间，最初规划硬盘分区时，留给分区表的空间决定了最多可以有多少个分区，IA-64版Windows限制最多有128个分区，这也是EFI标准规定的分区表的最小尺寸。另外，GPT分区磁盘有备份分区表来提高分区数据结构的完整性。

RAID 技术

磁盘阵列（Redundant Arrays of Independent Disks，RAID），独立冗余磁盘阵列之。原理是利用数组方式来作磁盘组，配合数据分散排列的设计，提升数据的安全性。

- RAID 0

  RAID 0是最早出现的RAID模式，需要2块以上的硬盘，可以提高整个磁盘的性能和吞吐量。

  RAID 0没有提供冗余或错误修复能力，其中一块硬盘损坏，所有数据将遗失。

- RAID 1

  RAID 1就是镜像，其原理为在主硬盘上存放数据的同时也在镜像硬盘上写一样的数据。

  当主硬盘（物理）损坏时，镜像硬盘则代替主硬盘的工作。因为有镜像硬盘做数据备份，所以RAID 1的数据安全性在所有的RAID级别上来说是最好的。

  但无论用多少磁盘做RAID 1，仅算一个磁盘的容量，是所有RAID中磁盘利用率最低的。

- RAID 2

  这是RAID 0的改良版，以汉明码（Hamming Code）的方式将数据进行编码后分区为独立的比特，并将数据分别写入硬盘中。因为在数据中加入了错误修正码（ECC，Error Correction Code），所以数据整体的容量会比原始数据大一些，RAID2最少要三台磁盘驱动器方能运作。

- RAID 3 采用Bit－interleaving（数据交错存储）技术，它需要通过编码再将数据比特分割后分别存在硬盘中，而将同比特检查后单独存在一个硬盘中，但由于数据内的比特分散在不同的硬盘上，因此就算要读取一小段数据资料都可能需要所有的硬盘进行工作，所

以这种规格比较适于读取大量数据时使用。

- RAID 4

它与RAID 3不同的是它在分区时是以区块为单位分别存在硬盘中，但每次的数据访问都必须从同比特检查的那个硬盘中取出对应的同比特数据进行核对，由于过于频繁的使用，所以对硬盘的损耗可能会提高。（块交织技术，Block interleaving）

RAID 2/3/4 在实际应用中很少使用

- RAID 5

  RAID Level 5是一种储存性能、数据安全和存储成本兼顾的存储解决方案。它使用的是Disk Striping（硬盘分区）技术。

  RAID 5至少需要三块硬盘，RAID 5不是对存储的数据进行备份，而是把数据和相对应的奇偶校验信息存储到组成RAID5的各个磁盘上，并且奇偶校验信息和相对应的数据分别存储于不同的磁盘上。

  RAID 5 允许一块硬盘损坏。

  实际容量 `Size = (N-1) * min(S1, S2, S3 ... SN)`

- RAID 6

  与RAID 5相比，RAID 6增加第二个独立的奇偶校验信息块。两个独立的奇偶系统使用不同的算法，数据的可靠性非常高，即使两块磁盘同时失效也不会影响数据的使用。

  RAID 6 至少需要4块硬盘。

  实际容量 `Size = (N-2) * min(S1, S2, S3 ... SN)`

- RAID 10/01（RAID 1+0，RAID 0+1）

  RAID 10是先镜射再分区数据，再将所有硬盘分为两组，视为是RAID 0的最低组合，然后将这两组各自视为RAID 1运作。

  RAID 01则是跟RAID 10的程序相反，是先分区再将数据镜射到两组硬盘。它将所有的硬盘分为两组，变成RAID 1的最低组合，而将两组硬盘各自视为RAID 0运作。

  当RAID 10有一个硬盘受损，其余硬盘会继续运作。RAID 01只要有一个硬盘受损，同组RAID 0的所有硬盘都会停止运作，只剩下其他组的硬盘运作，可靠性较低。如果以六个硬盘建RAID 01，镜射再用三个建RAID 0，那么坏一个硬盘便会有三个硬盘脱机。因此，RAID 10远较RAID 01常用，零售主板绝大部份支持RAID 0/1/5/10，但不支持RAID 01。

  RAID 10 至少需要4块硬盘，且硬盘数量必须为偶数。

常见文件系统

- Windows: FAT, FAT16, FAT32, NTFS
- Linux: ext2/3/4, btrfs, ZFS
- Mac OS X: HFS+

Linux文件权限

Linux文件采用10个标志位来表示文件权限，如下所示：

```
-rw-r--r--  1 skyline  staff    20B  1 27 10:34 1.txt
drwxr-xr-x   5 skyline  staff   170B 12 23 19:01 ABTableViewCell
```

第一个字符一般用来区分文件和目录，其中：

- d：表示是一个目录，事实上在ext2fs中，目录是一个特殊的文件。
- －：表示这是一个普通的文件。
- l: 表示这是一个符号链接文件，实际上它指向另一个文件。
- b、c：分别表示区块设备和其他的外围设备，是特殊类型的文件。
- s、p：这些文件关系到系统的数据结构和管道，通常很少见到。

第2～10个字符当中的每3个为一组，左边三个字符表示所有者权限，中间3个字符表示与所有者同一组的用户的权限，右边3个字符是其他用户的权限。

这三个一组共9个字符，代表的意义如下：

- r(Read，读取)：对文件而言，具有读取文件内容的权限；对目录来说，具有浏览目录的权限
- w(Write,写入)：对文件而言，具有新增、修改文件内容的权限；对目录来说，具有删除、移动目录内文件的权限。
- x(eXecute，执行)：对文件而言，具有执行文件的权限；对目录来说该用户具有进入目录的权限。

权限的掩码可以使用十进制数字表示：

- 如果可读，权限是二进制的100，十进制是4；
- 如果可写，权限是二进制的010，十进制是2；
- 如果可运行，权限是二进制的001，十进制是1；

- 具备多个权限，就把相应的 4、2、1 相加就可以了：

  若要 rwx 则 4+2+1=7 若要 rw- 则 4+2=6 若要 r-x 则 4+1=5 若要 r-- 则 =4 若要 -wx 则 2+1=3 若要 -w- 则 =2 若要 --x 则 =1 若要 --- 则 =0

默认的权限可用umask命令修改，用法非常简单，只需执行umask 777命令，便代表屏蔽所有的权限，因而之后建立的文件或目录，其权限都变成000，

依次类推。通常root帐号搭配umask命令的数值为022、027和 077，普通用户则是采用002，这样所产生的权限依次为755、750、700、775。

chmod命令

chmod命令非常重要，用于改变文件或目录的访问权限。用户用它控制文件或目录的访问权限。

该命令有两种用法。一种是包含字母和操作符表达式的文字设定法；另一种是包含数字的数字设定法。

1. 文字设定法

   chmod〔who〕〔+ | - | =〕〔mode〕文件名

   命令中各选项的含义为：

   操作对象who可是下述字母中的任一个或者它们的组合：

   - u 表示“用户（user）”，即文件或目录的所有者。
   - g 表示“同组（group）用户”，即与文件属主有相同组ID的所有用户。
   - o 表示“其他（others）用户”。
   - a 表示“所有（all）用户”。它是系统默认值。

   操作符号可以是：

   -
     - 添加某个权限。
   -
     - 取消某个权限。
   - = 赋予给定权限并取消其他所有权限（如果有的话）。

设置mode所表示的权限可用下述字母的任意组合：

- r 可读。
- w 可写。
- x 可执行。
- X 只有目标文件对某些用户是可执行的或该目标文件是目录时才追加x 属性。
- s 在文件执行时把进程的属主或组ID置为该文件的文件属主。方式“u＋s”设置文件的用户ID位，“g＋s”设置组ID位。
- t 保存程序的文本到交换设备上。
- u 与文件属主拥有一样的权限。
- g 与和文件属主同组的用户拥有一样的权限。
- o 与其他用户拥有一样的权限。

文件名：以空格分开的要改变权限的文件列表，支持通配符。

在一个命令行中可给出多个权限方式，其间用逗号隔开。例如：`chmod g+r，o+r example` 使同组和其他用户对文件example 有读权限。

1. 数字设定法

   直接使用数字表示的权限来更改：

   例： $ chmod 644 mm.txt

chgrp命令

功能：改变文件或目录所属的组。

语法：chgrp 〔选项〕 group filename

例：$ chgrp - R book /opt/local /book

改变/opt/local /book/及其子目录下的所有文件的属组为book。

chown命令

功能：更改某个文件或目录的属主和属组。这个命令也很常用。例如root用户把自己的一个文件拷贝给用户xu，为了让用户xu能够存取这个文件，root用户应该把这个文件的属主设为xu，否则，用户xu无法存取这个文件。

语法：chown 〔选项〕 用户或组 文件

说明：chown将指定文件的拥有者改为指定的用户或组。用户可以是用户名或用户ID。组可以是组名或组ID。文件是以空格分开的要改变权限的文件列表，支持通配符。

例：把文件shiyan.c的所有者改为wang。

```
chown wang shiyan.c
```

参考资料

- 操作系统中的磁盘调度算法
- 百度百科：磁盘阵列
- 维基百科：RAID
- Linux文件权限详解
- 修改Linux文件权限命令:chmod

事务的概念

事务的概念来自于两个独立的需求：并发数据库访问，系统错误恢复。

一个事务是可以被看作一个单元的一系列SQL语句的集合。

事务的特性（ACID）

- A, atomacity 原子性 事务必须是原子工作单元；对于其数据修改，要么全都执行，要么全都不执行。通常，与某个事务关联的操作具有共同的目标，并且是相互依赖的。如果系统只执行这些操作的一个子集，则可能会破坏事务的总体目标。原子性消除了系统处理操作子集的可能性。
- C, consistency 一致性

  事务将数据库从一种一致状态转变为下一种一致状态。也就是说，事务在完成时，必须使所有的数据都保持一致状态（各种 constraint 不被破坏）。
- I, isolation 隔离性 由并发事务所作的修改必须与任何其它并发事务所作的修改隔离。事务查看数据时数据所处的状态，要么是另一并发事务修改它之前的状态，要么是另一事务修改它之后的状态，事务不会查看中间状态的数据。换句话说，一个事务的影响在该事务提交前对其他事务都不可见。
- D, durability 持久性

  事务完成之后，它对于系统的影响是永久性的。该修改即使出现致命的系统故障也将一直保持。

事务的隔离级别

如果不对数据库进行并发控制，可能会产生异常情况：

1. 脏读(Dirty Read)

   当一个事务读取另一个事务尚未提交的修改时，产生脏读。

   同一事务内不是脏读。 一个事务开始读取了某行数据，但是另外一个事务已经更新了此数据但没有能够及时提交。这是相当危险的，因为很可能所有的操作都被回滚，也就是说读取出的数据其实是错误的。
2. 非重复读(Nonrepeatable Read) 一个事务对同一行数据重复读取两次，但是却得到了不同的结果。同一查询在同一事务中多次进行，由于其他提交事务所做的修改或删除，每次返回不同的结果集，此时发生非重复读。

3. 幻像读(Phantom Reads) 事务在操作过程中进行两次查询，第二次查询的结果包含了第一次查询中未出现的数据（这里并不要求两次查询的SQL语句相同）。这是因为在两次查询过程中有另外一个事务插入数据造成的。

   当对某行执行插入或删除操作，而该行属于某个事务正在读取的行的范围时，会发生幻像读问题。

4. 丢失修改(Lost Update)

   第一类：当两个事务更新相同的数据源，如果第一个事务被提交，第二个却被撤销，那么连同第一个事务做的更新也被撤销。

   第二类：有两个并发事务同时读取同一行数据，然后其中一个对它进行修改提交，而另一个也进行了修改提交。这就会造成第一次写操作失效。

为了兼顾并发效率和异常控制，在标准SQL规范中，定义了4个事务隔离级别，（ Oracle 和 SQL Server 对标准隔离级别有不同的实现 ）

1. 未提交读(Read Uncommitted)

   直译就是"读未提交"，意思就是即使一个更新语句没有提交，但是别的事务可以读到这个改变。

   Read Uncommitted允许脏读。

2. 已提交读(Read Committed)

   直译就是"读提交"，意思就是语句提交以后，即执行了 Commit 以后别的事务就能读到这个改变，只能读取到已经提交的数据。Oracle等多数数据库默认都是该级别。

   Read Commited 不允许脏读，但会出现非重复读。

1. 可重复读(Repeatable Read)：

   直译就是"可以重复读"，这是说在同一个事务里面先后执行同一个查询语句的时候，得到的结果是一样的。

   Repeatable Read 不允许脏读，不允许非重复读，但是会出现幻象读。

1. 串行读(Serializable)

   直译就是"序列化"，意思是说这个事务执行的时候不允许别的事务并发执行。完全串行化的读，每次读都需要获得表级共享锁，读写相互都会阻塞。

   Serializable 不允许不一致现象的出现。

事务隔离的实现——锁

1. 共享锁(S锁)

   用于只读操作(SELECT)，锁定共享的资源。共享锁不会阻止其他用户读，但是阻止其他的用户写和修改。

2. 更新锁(U锁)

   用于可更新的资源中。防止当多个会话在读取、锁定以及随后可能进行的资源更新时发生常见形式的死锁。

3. 独占锁(X锁，也叫排他锁)

   一次只能有一个独占锁用在一个资源上，并且阻止其他所有的锁包括共享缩。写是独占锁，可以有效的防止“脏读”。

Read Uncommited 如果一个事务已经开始写数据，则另外一个数据则不允许同时进行写操作，但允许其他事务读此行数据。该隔离级别可以通过“排他写锁”实现。

Read Committed 读取数据的事务允许其他事务继续访问该行数据，但是未提交的写事务将会禁止其他事务访问该行。可以通过“瞬间共享读锁”和“排他写锁”实现。

Repeatable Read 读取数据的事务将会禁止写事务（但允许读事务），写事务则禁止任何其他事务。可以通过“共享读锁”和“排他写锁”实现。

Serializable 读加共享锁，写加排他锁，读写互斥。

参考资料

- 百度百科：数据库事务
- 数据库事务隔离级别与锁
- SQL SERVER的锁机制（四）——概述（各种事务隔离级别发生的影响）
- 数据库锁
- 关于数据库事务、隔离级别、锁的理解与整理
- Innodb中的事务隔离级别和锁的关系

数据库创建索引能够大大提高系统的性能。
第一，通过创建唯一性的索引，可以保证数据库表中每一行数据的唯一性。
第二，可以大大加快数据的检索速度，这也使创建索引的最主要的原因。
第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。
第四，在使用分组和排序子句进行数据检索时，同样可以显著的减少查询中查询中分组和排序的时间。
第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

增加索引也有许多不利的方面。
第一，创建索引和维护索引需要消耗时间，这种时间随着数量的增加而增加。
第二，索引需要占物理空间，除了数据表占据数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要额空间就会更大。
第三，当对表中的数据进行增加，删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

应该对如下的列建立索引

1. 在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构。
2. 在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度。
3. 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的。
4. 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间。
5. 在经常使用在where子句中的列上面创建索引，加快条件的判断速度。

有些列不应该创建索引

1. 在查询中很少使用或者作为参考的列不应该创建索引。
2. 对于那些只有很少数据值的列也不应该增加索引（比如性别，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度）。
3. 对于那些定义为text，image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。
4. 当修改性能远远大于检索性能时，不应该创建索引，因为修改性能和检索性能是矛盾的。

创建索引的方法：直接创建和间接创建（在表中定义主键约束或者唯一性约束时，同时也创建了索引）。
索引的特征：
唯一性索引和复合索引。唯一性索引保证在索引列中的全部数据是唯一的，不会包含冗余数据。复合索引就是一个索引创建在两个列或者多个列上。可以减少一在一个表中所创建的索引数量。

词法分析器

语法分析器

语义分析及中间代码生成

代码优化

代码生成

面向对象的基本特征

面向对象的三个基本特征是：封装、继承、多态

- 封装

  封装最好理解了。封装是面向对象的特征之一，是对象和类概念的主要特性。封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

- 继承

  继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。

  要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。

- 多态性

  多态性（polymorphisn）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。

  实现多态，有两种方式，覆盖和重载。覆盖和重载的区别在于，覆盖在运行时决定，重载是在编译时决定。并且覆盖和重载的机制不同，例如在 Java 中，重载方法的签名必须不同于原先方法的，但对于覆盖签名必须相同。

参考资料

1. 面向对象的三个基本特征

- https://progit.org/
- http://think-like-a-git.net/

iOS 工程师技能树

http://segmentfault.com/a/1190000002946644

Objective-C

- Objective-C语言基础
- library,framework的制作
- Runtime 编程
- LLVM 原理和调优

操作系统

- iOS内存管理和调优
- iOS的文件系统和沙盒机制
- iOS多线程编程（Thread,GCD,NSOperation）
- iOS网络和服务器编程（NSURLConnection,NSURLSession）
- iOS系统的各种安全机制

网络编程

- iOS网络发送机制调整和优化（NSURLSession）
- Socket编程
- 网络传输中的各种保障
- 对传输协议的调整优化

数据库&持久化方案

- 常规持久化方案（Keychain,NSUserDefaults,Sqlite,CoreData）
- 数据库的使用和设计（Sqlite）
- 数据结构优化，Sql调优

图形图像编程

- UIKit,Core Animation和Core Text的绘制
- Core Graphics, Quartz 2D, Media Player, AV Foundation
- OpenGL ES, GLKit, SpriteKit, SceneKit, Metal

数据结构 & 算法

- 基本的算法和数据结构（排序搜索算法，数组，队列）
- 较复杂数据结构的灵活应用（二叉树，图等）
- 复杂的专项算法（图像识别算法，拓扑定位等等）

安全方案

- 本地数据存储安全（Keychain）
- 授权和身份验证
- 传输安全（对称, 非对称, SSL）
- App代码安全

业务能力

- 一般性业务功能需求分析及实现
- 重要业务模块的需求分析及实现
- 中小规模产品的架构，系统设计和实现
- 大规模产品或产品线的架构，系统设计和实现
- 平台级产品的架构，系统设计和实现

类方法

OC中类的方法只有实例方法和静态方法两种：

```
@interface Controller : NSObject

+ (void)thisIsAStaticMethod; // 静态方法

- (void)thisIsAnInstanceMethod; // 实例方法

@end
```

OC 中的方法只要声明在 @interface里，就可以认为都是公有的。实际上，OC 没有像 Java，C++ 中的那种绝对的私有及保护成员方法，仅仅可以对调用者隐藏某些方法。

声明和实现都写在 @implementation 里的方法，类的外部是看不到的。

可以使用 Category 来实现私有方法：

```
// AClass.h
@interface AClass : NSObject

-(void)sayHello;

@end

// AClass.m
@interface AClass (private)

-(void)privateSayHello;

@end

@implementation AClass

-(void)sayHello {
    [self privateSayHello];
}

-(void)privateSayHello {
    NSLog(@"Private Hello");
}
```

使用这种方法时，外部就不能直接调用到 `privateSayHello` 方法。

注意在上面的代码里面，当我们想通过 Category 来进行方法隐藏的时候，我们可以把实现放在主 implementation 里。当我们想扩展别的不能获取到源代码的类，或者想把不同 Category 的实现分开，可以新建 `<ClassName>+CategoryName.m` 文件，在里面进行实现：

```
#import "SystemClass+CategoryName.h"

@implementation SystemClass ( CategoryName )
// method definitions
@end
```

也可以使用 Extension 来实现私有方法：

```
// AClass.h 与上面相同

// AClass.m
@interface AClass()

-(void)privateSayHello;

@end

@implementation AClass

-(void)sayHello {
    [self privateSayHello];
}

-(void)privateSayHello {
    NSLog(@"Private Hello");
}

@end
```

与使用 Category 类似，由于声明隐藏在 .m 中，调用者无法看到其声明，也就无法调用 `privateSayHello` 这个方法，会引发编译错误。

关于 Category 和 Extension 的一些区别，在这里。

类变量

苹果推荐在现代 Objective-C 中使用 `@property` 来实现成员变量：

```
@interface AClass : NSObject

@property (nonatomic, copy) NSString *name;

@end
```

使用 `@property` 声明的变量可以使用 `实例名.变量名` 来获取和修改。

`@property` 可以看做是一种语法糖，在 MRC 下，使用 `@property` 可以看成实现了下面的代码：

```
// AClass.h
@interface AClass : NSObject{
@public
    NSString *_name;
}

-(NSString*)name;
-(void)setName:(NSString*)newName;
@end

// AClass.m
@implementation AClass

-(NSString*)name{
    return _name;
}

-(void)setName:(NSString *)name{
    if (_name != name) {
        [_name release];
        _name = [name copy];
    }
}
@end
```

也就是说， `@property` 会自动生成 getter 和 setter， 同时进行自动内存管理。

`@property` 的属性可以有以下几种：

- readwrite 是可读可写特性；需要生成 getter 方法和 setter 方法
- readonly 是只读特性，只会生成 getter 方法 不会生成 setter 方法，不希望属性在类外改变时使用
- assign 是赋值特性，setter 方法将传入参数赋值给实例变量；仅设置变量时；
- retain 表示持有特性，setter 方法将传入参数先保留，再赋值，传入参数的 retain count 会+1;
- copy 表示拷贝特性，setter 方法将传入对象复制一份；需要完全一份新的变量时。
- nonatomic 和 atomic ，决定编译器生成的 setter getter是否是原子操作。 atomic 表示使用原子操作，可以在一定程度上保证线程安全。一般推荐使用 nonatomic ，因为 nonatomic 编译出的代码更快

默认的 `@property` 是 readwrite，assign，atomic。

同时，我们还可以使用自己定义 accessor 的名字：

```
@property (getter=isFinished) BOOL finished;
```

这种情况下，编译器生成的 getter 方法名为 `isFinished` ，而不是 `finished` 。

@synthesize 和 @dynamic

对于现代 OC 来说，在使用 `@property` 时， 编译器默认会进行自动 synthesize，生成 getter 和 setter，同时把 ivar 和属性绑定起来：

```
/// 现代 OC 不再需要手动进行下面的声明，编译器会自动处理
@synthesize propertyName = _propertyName
```

不需要我们写任何代码，就可以直接使用 getter 和 setter 了。

然而并不是所有情况下编译器都会进行自动 synthesize，具体由下面几种：

- 可读写(readwrite)属性实现了自己的 getter 和 setter
- 只读(readonly)属性实现了自己的 getter
- 使用 `@dynamic`，显式表示不希望编译器生成 getter 和 setter
- protocol 中定义的属性，编译器不会自动 synthesize，需要手动写
- 当重载父类中的属性时，也必须手动写 synthesize

类的扩展——Protocol, Category 和 Extension

Protocol

OC是单继承的，OC中的类可以实现多个 protocol 来实现类似 C++ 中多重继承的效果。

Protocol 类似 Java 中的 interface，定义了一个方法列表，这个方法列表中的方法可以使用 `@required`，`@optional` 标注，以表示该方法是否是客户类必须要实现的方法。 一个 protocol 可以继承其他的 protocol 。

```
@protocol TestProtocol<NSObject> // NSObject也是一个 Protocol，这里即继承 NSObject 里的方法
-(void)print;
@end

@interface B : NSObject<TestProtocol>
-(void)print; // 默认方法是 @required 的，即必须实现
@end
```

Delegate（委托）是 Cocoa 中常见的一种设计模式，其实现依赖于 protocol 这个语言特性。

含有 property 的 Protocol

上面提到过，当 Protocol 中含有 property 时，编译器是不会进行自动 synthesize 的，需要手动处理：

```
@class ExampleClass;

@protocol ExampleProtocol

@required

@property (nonatomic, retain) ExampleClass *item;

@end
```

在实现这个 Protocol 的时候，要么再次声明 property：

```
@interface MyObject : NSObject <ExampleProtocol>

@property (nonatomic, retain) ExampleClass *item;

@end
```

要么进行手动 synthesize：

```
@interface MyObject : NSObject <ExampleProtocol>
@end

@implementation MyObject
@synthesize item;

@end
```

工程自带的 `AppDelegate` 使用了前一种方法，`UIApplicationDelegate` protocol 当中定义了 `window` 属性：

```
@property (nonatomic, retain) UIWindow *window NS_AVAILABLE_IOS(5_0);
```

在 `AppDelegate.h` 中我们可以看到再次对 `windows` 进行了声明：

```
@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (nonatomic, strong) UIWindow *window;

@end
```

Category

Category 是一种很灵活的扩展原有类的机制，使用 Category 不需要访问原有类的代码，也无需继承。Category提供了一种简单的方式，来实现类的相关方法的模块化，把不同的类方法分配到不同的类文件中。

Category 常见的使用方法如下：

```
// SomeClass.h
@interface SomeClass : NSObject{
}
-(void)print;
@end

// SomeClass+Hello.h
#import "SomeClass.h"

@interface SomeClass (Hello)
-(void)hello;
@end

// 实现
#import "SomeClass+Hello.h"
@implementationSomeClass (Hello)
-(void)hello{
    NSLog(@"name：%@ ", @"Jacky");
}
@end
```

在使用 Category 时需要注意的一点是，如果有多个命名 Category 均实现了同一个方法（即出现了命名冲突），那么这些方法在运行时只有一个会被调用，具体哪个会被调用是不确定的。因此在给已有的类（特别是 Cocoa 类）添加 Category 时，推荐的函数命名方法是加上前缀：

```
@interface NSSortDescriptor (XYZAdditions)
+ (id)xyz_sortDescriptorWithKey:(NSString *)key ascending:(BOOL)ascending;
@end
```

Extension

Extension 可以认为是一种匿名的 Category， Extension 与 Category 有如下几点显著的区别：

1. 使用 Extension 必须有原有类的源码
2. Extension 声明的方法必须在类的主 @implementation 区间内实现，可以避免使用有名 Category 带来的多个不必要的 implementation 段。
3. Extension 可以在类中添加新的属性和实例变量，Category 不可以（注：在 Category 中实际上可以通过运行时添加新的属性，下面会讲到）
4. ~~Extension 里添加的方法必须要有实现（没有实现编译器会给出警告）~~

> 注：现代 ObjC 中 Extension 和 Category 中声明的方法如果没有实现编译器都会给出警告。

下面是一个 Extension 的例子：

```
@interface MyClass : NSObject
- (float)value;
@end

@interface MyClass () { // 注意此处扩展的写法
    float value;
}
- (void)setValue:(float)newValue;
@end

@implementation MyClass

- (float)value {
    return value;
}

- (void)setValue:(float)newValue {
    value = newValue;
}
@end
```

Extension 很常见的用法，是用来给类添加私有的变量和方法，用于在类的内部使用。例如在 interface 中定义为 `readonly` 类型的属性，在实现中添加 extension，将其重新定义为 `readwrite`，这样我们在类的内部就可以直接修改它的值，然而外部依然不能调用 `setter` 方法来修改。示例代码如下（来自苹果官方文档）：

`XYZPerson.h`

```
@interface XYZPerson : NSObject
...
@property (readonly) NSString *uniqueIdentifier;

@end
```

`XYZPerson.m`

```
@interface XYZPerson ()
@property (readwrite) NSString *uniqueIdentifier;
@end

@implementation XYZPerson
...
@end
```

如何给已有的类添加属性

首先强调一下上面例子中所展示的，Extension 可以给类添加属性，编译器会自动生成 getter，setter 和 ivar。 Category 并不支持这些。如果使用 Category 的话，类似下面这样：

```
@interface XYZPerson (UDID)
@property (readwrite) NSString *uniqueIdentifier;
@end

@implementation XYZPerson (UDID)
...
@end
```

尽管编译可以通过，但是当真正使用 `uniqueIdentifier` 时直接会导致程序崩溃。

如果我们手动去 synthesize 呢？像下面这样：

```
@implementation XYZPerson (UDID)
@synthesize uniqueIdentifier;
...
@end
```

然而这样做的话，代码直接报编译错误了：

`@synthesize not allowed in a category's implementation`

看来这条路是彻底走不通了。

不过我们还有别的方法，想通过 Category 添加属性的话，可以通过 Runtime 当中提供的 associated object 特性。NSHipster 的 这篇文章 展示了具体的做法。

如何在类中添加全局变量

有些时候我们需要在类中添加某个在类中全局可用的变量，为了避免污染作用域，一个比较好的做法是在 .m 文件中使用 static 变量：

```
static NSOperationQueue * _personOperationQueue = nil;

@implementation XYZPerson
...
@end
```

由于 static 变量在编译期就是确定的，因此对于 NSObject 对象来说，初始化的值只能是 nil。如何进行类似 init 的初始化呢？可以通过重载 initialize 方法来做：

```
@implementation XYZPerson
- (void)initialize {
    if (!_personOperationQueue) {
        _personOperationQueue = [[NSOperationQueue alloc] init];
    }
}
@end
```

为什么这里要判断是否为 nil 呢？因为 `initialize` 方法可能会调用多次，后面会提到。

如果是在 Category 中想声明全局变量呢？当然也可以通过 initialize，不过除非必须的情况下，并不推荐在 Category 当中进行方法重载。

有一种方法是声明 static 函数，下面的代码来自 AFNetworking，声明了一个当前文件范围可用的队列：

```
static dispatch_queue_t url_session_manager_creation_queue() {
    static dispatch_queue_t af_url_session_manager_creation_queue;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        af_url_session_manager_creation_queue = dispatch_queue_create("com.alamofire.n
etworking.session.manager.creation", DISPATCH_QUEUE_SERIAL);
    });

    return af_url_session_manager_creation_queue;
}
```

下面介绍一个有点黑魔法的方法，除了上面两种方法之外，我们还可以通过编译器的 `__attribute__` 特性来实现初始化：

```
__attribute__((constructor))
static void initialize_Queue() {
    _personOperationQueue = [[NSOperationQueue alloc] init];
}

@implementation XYZPerson (Operation)

@end
```

类的导入

导入类可以使用 `#include` ， `#import` 和 `@class` 三种方法，其区别如下：

- `#import` 是Objective-C导入头文件的关键字， `#include` 是C/C++导入头文件的关键字
- 使用 `#import` 头文件会自动只导入一次，不会重复导入，相当于 `#include` 和 `#pragma once` ；
- `@class` 告诉编译器需要知道某个类的声明，可以解决头文件的相互包含问题；

`@class` 是放在interface中的，只是在引用一个类，将这个被引用类作为一个类型使用。在实现文件中，如果需要引用到被引用类的实体变量或者方法时，还需要使用 `#import` 方式引入被引用类。

类的初始化

Objective-C 是建立在 Runtime 基础上的语言，类也不例外。OC 中类是初始化也是动态的。在 OC 中绝大部分类都继承自 `NSObject` ，它有两个非常特殊的类方法 `load` 和 `initilize` ，用于类的初始化

+load

+load 方法是当类或分类被添加到 Objective-C runtime 时被调用的，实现这个方法可以让我们在类加载的时候执行一些类相关的行为。子类的 +load 方法会在它的所有父类的 +load 方法之后执行，而分类的 +load 方法会在它的主类的 +load 方法之后执行。但是不同的类之间

的 +load 方法的调用顺序是不确定的。

load 方法不会被类自动继承, 每一个类中的 load 方法都不需要像 viewDidLoad 方法一样调用父类的方法。子类、父类和分类中的 +load 方法的实现是被区别对待的。也就是说如果子类没有实现 +load 方法，那么当它被加载时 runtime 是不会去调用父类的 +load 方法的[1]。同理，当一个类和它的分类都实现了 +load 方法时，两个方法都会被调用。因此，我们常常可以利用这个特性做一些“邪恶”的事情，比如说方法混淆（Method Swizzling）。FDTemplateLayoutCell 中就使用了这个方法，见这里。

+initialize

+initialize 方法是在类或它的子类收到第一条消息之前被调用的，这里所指的消息包括实例方法和类方法的调用。也就是说 +initialize 方法是以懒加载的方式被调用的，如果程序一直没有给某个类或它的子类发送消息，那么这个类的 +initialize 方法是永远不会被调用的。

+initialize 方法的调用与普通方法的调用是一样的，走的都是发送消息的流程。换言之，如果子类没有实现 +initialize 方法，那么继承自父类的实现会被调用；如果一个类的分类实现了 +initialize 方法，那么就会对这个类中的实现造成覆盖。

注解

-

1.举一个例子：有一个 Father 类，实现了 load 方法，打印类名，一个 Son 类继承自前者，没有实现 load 方法。实例出一个 Son 的对象时，结果是会输出父类的名字。但这个例子与之前的结论并不矛盾，这里说的是父类先被加载了，所以调用了父类的 load 方法，而子类被加载时没有调用父类的 load 方法。 暂时没找到例子可以严格的证明此前的结论，所以还是去看源码吧。↩

参考资料

- iOS开发基础面试题系列
- 10个Objective-C基础面试题，iOS面试必备
- Objective-C中“私有方法”的实现"
- Objective-C中@property详解
- Objective-C中的protocol和delegate
- Objective-C——消息，Category 与 Protocol
- 深入理解Objective-C中的@class
- Objective-C +load vs +initialize
- 深入理解Objective-C：Category
- https://stackoverflow.com/questions/19784454/when-should-i-use-synthesize-explicitly
- http://www.fantageek.com/blog/2014/07/13/property-in-protocol/
- http://www.friday.com/bbum/2009/09/06/iniailize-can-be-executed-multiple-times-load-

not-so-much/

Block 基础

Block 语法

Block 可以认为是一种匿名函数，使用如下语法声明一个 Block 类型：

```
return_type (^block_name)(parameters)
```

例如：

```
double (^multiplyTwoValues)(double, double);
```

Block 字面值的写法如下：

```
^ (double firstValue, double secondValue) {
    return firstValue * secondValue;
}
```

上面的写法省略了返回值的类型，也可以显式地指出返回值类型。

声明并且定义完一个Block之后，便可以像使用函数一样使用它：

```
double (^multiplyTwoValues)(double, double) =
                          ^(double firstValue, double secondValue) {
                              return firstValue * secondValue;
                          };
double result = multiplyTwoValues(2,4);

NSLog(@"The result is %f", result);
```

同时，Block 也是一种 Objective-C 对象，可以用于赋值，当做参数传递，也可以放入 NSArray 和 NSDictionary 中。

注意：当用于函数参数时，Block 应该放在参数列表的最后一个。

Bonus： 由于 Block 的语法是如此的晦涩难记，以至于出现了 fuckingblocksyntax 这样的网站专门用于记录 block 的语法，翻译并摘录如下：

作为变量:

```
returnType (^blockName)(parameterTypes) = ^returnType(parameters) {...};
```

作为属性:

```
@property (nonatomic, copy) returnType (^blockName)(parameterTypes);
```

作为函数声明中的参数:

```
- (void)someMethodThatTakesABlock:(returnType (^)(parameterTypes))blockName;
```

作为函数调用中的参数:

```
[someObject someMethodThatTakesABlock:^returnType (parameters) {...}];
```

作为 typedef:

```
typedef returnType (^TypeName)(parameterTypes);
TypeName blockName = ^returnType(parameters) {...};
```

Block 可以捕获外部变量

Block 可以捕获来自外部作用域的变量，这是Block一个很强大的特性。

```
- (void)testMethod {
    int anInteger = 42;
    void (^testBlock)(void) = ^{
        NSLog(@"Integer is: %i", anInteger);
    };
    testBlock();
}
```

默认情况下，Block 中捕获的到变量是不能修改的，如果想修改，需要使用 `__block` 来声明：

```
__block int anInteger = 42;
```

对于 id 类型的变量，在 MRC 情况下，使用 `__block id x` 不会 retain 变量，而在 ARC 情况下则会对变量进行 retain（即和其他捕获的变量相同）。如果不想在 block 中进行 retain 可以使用 `__unsafe_unretained __block id x`，不过这样可能会导致野指针出现。更好的办法是使用 `__weak` 的临时变量：

```
MyViewController *myController = [[MyViewController alloc] init…];
// ...
MyViewController * __weak weakMyViewController = myController;
myController.completionHandler =  ^(NSInteger result) {
    [weakMyViewController dismissViewControllerAnimated:YES completion:nil];
};
```

或者把使用 `__block` 修饰的变量设为 nil，以打破引用循环：

```
MyViewController * __block myController = [[MyViewController alloc] init…];
// ...
myController.completionHandler =  ^(NSInteger result) {
    [myController dismissViewControllerAnimated:YES completion:nil];
    myController = nil;
};
```

Block 进阶

使用 Block 时的注意事项

在非 ARC 的情况下，对于 block 类型的属性应该使用 `copy` ，因为 block 需要维持其作用域中捕获的变量。在 ARC 中编译器会自动对 block 进行 copy 操作，因此使用 `strong` 或者 `copy` 都可以，没有什么区别，但是苹果仍然建议使用 `copy` 来指明编译器的行为。

block 在捕获外部变量的时候，会保持一个强引用，当在 block 中捕获 `self` 时，由于对象会对 block 进行 `copy` ，于是便形成了强引用循环：

```
@interface XYZBlockKeeper : NSObject
@property (copy) void (^block)(void);
@end
```

```
@implementation XYZBlockKeeper
- (void)configureBlock {
    self.block = ^{
        [self doSomething];     // capturing a strong reference to self
                                // creates a strong reference cycle
    };
}
...
@end
```

为了避免强引用循环，最好捕获一个 `self` 的弱引用：

```
- (void)configureBlock {
    XYZBlockKeeper * __weak weakSelf = self;
    self.block = ^{
        [weakSelf doSomething];    // capture the weak reference
                                   // to avoid the reference cycle
    }
}
```

使用弱引用会带来另一个问题， `weakSelf` 有可能会为 nil，如果多次调用 `weakSelf` 的方法，有可能在 block 执行过程中 `weakSelf` 变为 nil。因此需要在 block 中将 `weakSelf` “强化“

```
__weak __typeof__(self) weakSelf = self;
NSBlockOperation *op = [[[NSBlockOperation alloc] init] autorelease];
[ op addExecutionBlock:^ {
    __strong __typeof__(self) strongSelf = weakSelf;
    [strongSelf doSomething];
    [strongSelf doMoreThing];
} ];
[someOperationQueue addOperation:op];
```

`__strong` 这一句在执行的时候，如果 WeakSelf 还没有变成 nil，那么就会 retain self，让 self 在 block 执行期间不会变为 nil。这样上面的 `doSomething` 和 `doMoreThing` 要么全执行成功，要么全失败，不会出现一个成功一个失败，即执行到中间 `self` 变成 nil 的情况。

Bonus

很多文章对于 weakSelf 的解释中并没有详细说，为什么有可能 block 执行的过程当中 weakSelf 变为 nil，这就涉及到 weak 本身的机制了。weak 置 nil 的操作发生在 dealloc 中，苹果在 TN2109 - The Deallocation Problem 中指出，最后一个持有 object 的对象被释放的时候，会触发对象的 dealloc，而这个持有者的释放操作就不一定保证发生在哪个线程了。因此 block 执行的过程中 weakSelf 有可能在另外的线程中被置为 nil。

Block 在堆上还是在栈上？

首先要指出，Block 在非 ARC 和 ARC 两种环境下的内存机制差别很大。

在 MRC 下，Block 默认是分配在栈上的，除非进行显式的 copy：

```
__block int val = 10;
blk stackBlock = ^{NSLog(@"val = %d", ++val);};
NSLog(@"stackBlock: %@", stackBlock); // stackBlock: <__NSStackBlock__: 0xbfffdb28>

tempBlock = [stackBlock copy];
NSLog(@"tempBlock: %@", tempBlock);  // tempBlock: <__NSMallocBlock__: 0x756bf20>
```

想把 Block 用作返回值的时候，也要加入 `copy` 和 `autorelease` ：

```
- (blk)myTestBlock {
    __block int val = 10;
    blk stackBlock = ^{NSLog(@"val = %d", ++val);};
    return [[stackBlock copy] autorelease];
}
```

在 ARC 环境下，Block 使用简化了很多，同时 ARC 也更加倾向于把 Block 放到堆上：

```
__block int val = 10;
__strong blk strongPointerBlock = ^{NSLog(@"val = %d", ++val);};
NSLog(@"strongPointerBlock: %@", strongPointerBlock); // strongPointerBlock: <__NSMall
ocBlock__: 0x7625120>

__weak blk weakPointerBlock = ^{NSLog(@"val = %d", ++val);};
NSLog(@"weakPointerBlock: %@", weakPointerBlock); // weakPointerBlock: <__NSStackBlock
__: 0xbfffdb30>

NSLog(@"mallocBlock: %@", [weakPointerBlock copy]); // mallocBlock: <__NSMallocBlock__
: 0x714ce60>

NSLog(@"test %@", ^{NSLog(@"val = %d", ++val);}); // test <__NSStackBlock__: 0xbfffdb1
8>
```

可以看到只有显式的 `__weak` 以及纯匿名 Block 是放到栈上的，赋值给 `__strong` 指针（也就是默认赋值）都会导致在堆上创建 Block。

对于把 Block 作为函数返回值的情况，ARC 也能自动处理：

```
- (__unsafe_unretained blk) blockTest {
    int val = 11;
    return ^{NSLog(@"val = %d", val);};
}

NSLog(@"block return from function: %@", [self blockTest]); // block return from funct
ion: <__NSMallocBlock__: 0x7685640>
```

PS：经过上面的讨论，可以发现巧神的这篇博客中认为在 ARC 情况下不再有 `NSConcreteStackBlock`，其实是不完全准确的。

参考资料

- https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html#//apple_ref/doc/uid/TP40011210-CH8-SW16
- http://blog.waterworld.com.hk/post/block-weakself-strongself
- https://stackoverflow.com/questions/17384599/why-are-block-variables-not-retained-in-non-arc-environments
- https://stackoverflow.com/questions/2746197/dealloc-on-background-thread/24410372#24410372
- http://www.cnblogs.com/biosli/archive/2013/05/29/iOS_Objective-C_Block.html

Objective-C Runtime

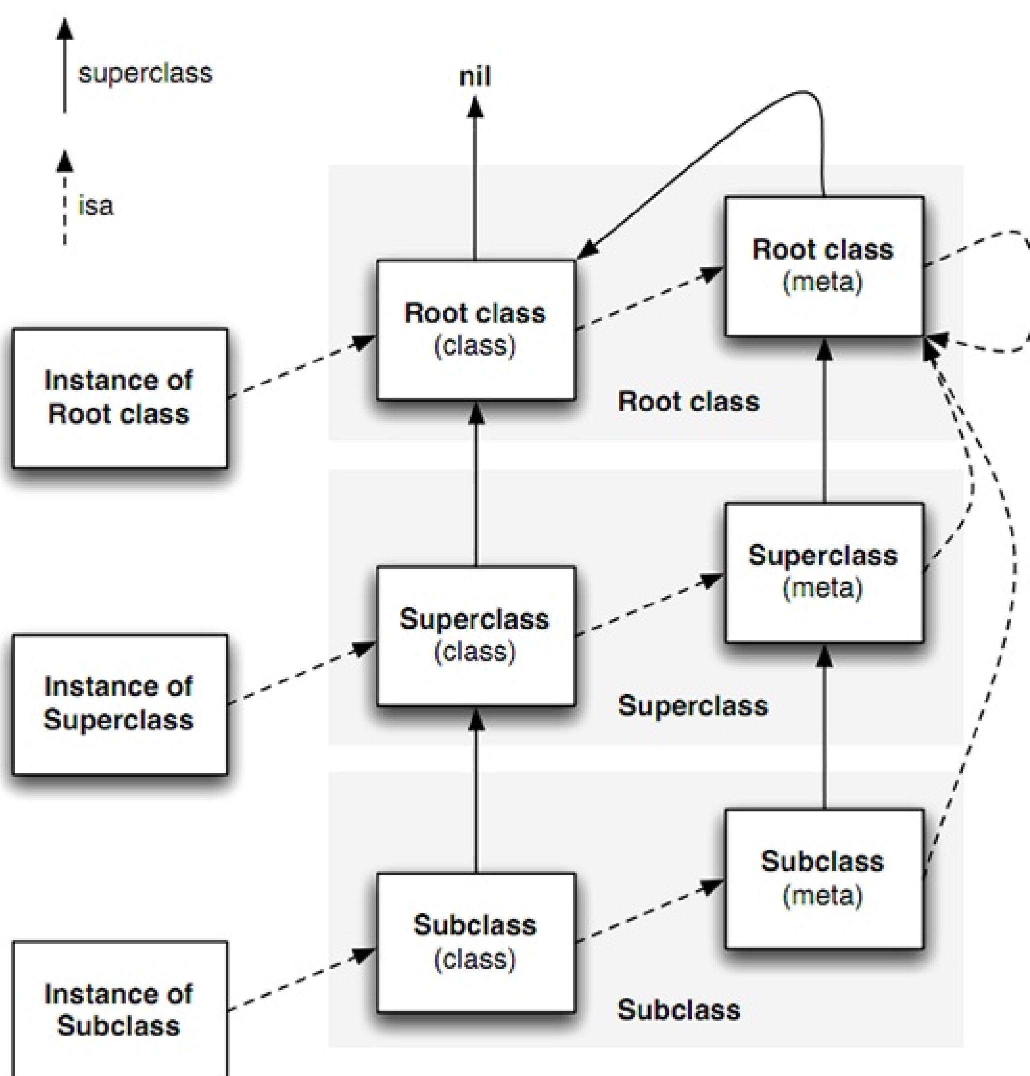
Runtime 是什么？

Runtime 是 Objective-C 区别于 C 语言这样的静态语言的一个非常重要的特性。对于 C 语言，函数的调用会在编译期就已经决定好，在编译完成后直接顺序执行。但是 OC 是一门动态语言，函数调用变成了消息发送，在编译期不能知道要调用哪个函数。所以 Runtime 无非就是去解决如何在运行时期找到调用方法这样的问题。

对于实例变量有如下的思路：

> instance -> class -> method -> SEL -> IMP -> 实现函数

实例对象中存放 isa 指针以及实例变量，有 isa 指针可以找到实例对象所属的类对象 (类也是对象，面向对象中一切都是对象)，类中存放着实例方法列表，在这个方法列表中 SEL 作为 key，IMP 作为 value。 在编译时期，根据方法名字会生成一个唯一的 Int 标识，这个标识就是 SEL。IMP 其实就是函数指针 指向了最终的函数实现。整个 Runtime 的核心就是 objc_msgSend 函数，通过给类发送 SEL 以传递消息，找到匹配的 IMP 再获取最终的实现。如下的这张图描述了对象的内存布局。



类中的 super_class 指针可以追溯整个继承链。向一个对象发送消息时，Runtime 会根据实例对象的 isa 指针找到其所属的类，并自底向上直至根类(NSObject)中 去寻找 SEL 所对应的方法，找到后就运行整个方法。

metaClass是元类，也有 isa 指针、super_class 指针。其中保存了类方法列表。

如下是 objc/runtime.h 中定义的类的结构:

```
struct objc_class {
    Class isa  OBJC_ISA_AVAILABILITY;

#if !__OBJC2__
    Class super_class                                        OBJC2_UNAVAILABLE;
    const char *name                                         OBJC2_UNAVAILABLE;
    long version                                             OBJC2_UNAVAILABLE;
    long info                                                OBJC2_UNAVAILABLE;
    long instance_size                                       OBJC2_UNAVAILABLE;
    struct objc_ivar_list *ivars                             OBJC2_UNAVAILABLE; // 成员变量地址列表
    struct objc_method_list **methodLists                    OBJC2_UNAVAILABLE; // 方法地址列表
    struct objc_cache *cache                                 OBJC2_UNAVAILABLE; // 缓存最近使用的方法地址，以避免多次在方法地址列表中查询，提升效率
    struct objc_protocol_list *protocols                     OBJC2_UNAVAILABLE; // 遵循的协议列表
#endif

} OBJC2_UNAVAILABLE;
/* Use `Class` instead of `struct objc_class *` */
```

SEL 与 IMP

SEL 可以将其理解为方法的 ID. 结构如下：

```
typedef struct objc_selector *SEL;

struct objc_selector {
    char *name;                       OBJC2_UNAVAILABLE;
    char *types;                      OBJC2_UNAVAILABLE;
};
```

IMP 可以理解为函数指针，指向了最终的实现。

SEL 与 IMP 的关系非常类似于 HashTable 中 key 与 value 的关系。OC 中不支持函数重载的原因就是因为一个类的方法列表中不能存在两个相同的 SEL 。但是多个方法却可以在不同的类中有一个相同的 SEL，不同类的实例对象执行相同的 SEL 时，会在各自的方法列表中去根据 SEL 去寻找自己对应的IMP。这使得OC可以支持函数重写。

消息传递机制

- objc_msgSend函数的消息处理过程
- 不涵盖消息cache机制
- 需要对Objective-C runtime有一定的了解

如下用于描述 objc_msgSend 函数的调用流程：

- 1.检测 SEL 是否应该被忽略
- 2.检测发送的 target 是否为 nil ，如果是则忽略该消息
- 3.
    - 当调用实例方法时，通过 isa 指针找到实例对应的 class 并且在其中的缓存方法列表

    以及方法列表中进行查询，如果找不到则根据 super_class 指针在父类中查询，直至根类(NSObject 或 NSProxy).
  - 当调用类方法时，通过 isa 指针找到实例对应的 metaclass 并且在其中的缓存方法列表以及方法列表中进行查询，如果找不到则根据 super_class 指针在父类中查询，直至根类(NSObject 或 NSProxy). (根据此前的开篇中的图，Root Meta Class 还是有根类的。)
- 如果还没找到则进入消息动态解析过程。

由于苹果对*OC 2.0 Runtime*的具体实现细节未完全开源，本节所引用源代码大部分来自*OC 1.0*，如有错误，敬请更正

当一个对象 sender 调用代码 `[receiver message];` 的时候，实际上是调用了runtime 的 `objc_msgSend` 函数，所以OC的方法调用并不像C函数一样能按照地址直接取用，而是经过了一系列的过程。这样的机制使得 runtime 可以在接收到消息后对消息进行特殊处理，这才使OC的一些特性譬如：给 nil 发送消息不崩溃，给类动态添加方法和消息转发等成为可能。也正因为每一次调用方法的时候实际上是调用了一些 runtime 的消息处理函数，OC的方法调用相对于C来说会相对较慢，但 OC 也通过引入 cache 机制来很大程度上的克服了这个缺点。下面我们就从一个对象 sender 调用代码 `[receiver message];` 这个情景开始，了解消息传递的过程。

首先这行代码会被改写成 `objc_msgSend(self, _cmd);` ，这是一个runtime的函数，其原型为：

`id objc_msgSend(id self, SEL op, ...)`

self与_cmd是两个编译器会自动添加的隐藏参数，self是一个指向接收对象的指针，_cmd为方法选择器。这个函数的实现为汇编版本，苹果开源的项目中共有6种对不同平台的汇编实现，本节选取其在x86_64实现的文件objc-msg-x86_64.s

```
#objc-msg-x86_64.s#
    ENTRY    _objc_msgSend
    // ...
    GetIsaFast NORMAL        // r11 = self->isa
    CacheLookup NORMAL        // calls IMP on success
    // ...
// cache miss: go search the method lists
LCacheMiss:
    // isa still in r11
    MethodTableLookup %a1, %a2    // r11 = IMP
    cmp    %r11, %r11        // set eq (nonstret) for forwarding
    jmp    *%r11            // goto *imp
    END_ENTRY    _objc_msgSend
```

可以看到其调用了 `GetIsaFast` ，由于self是id类型，而id的原型为 `struct objc_object *id;` ，所以需要通过id的isa指针获取其所属的类对象，之后调用 `CacheLookup` 在获取到的类中根据传入的_cmd查找对应方法实现的IMP指针。这两个函数的实现均在同一个文件下，因为暂时我还不了解cache的机制，所以这部分先不深入讨论。CacheLookup函数在命中后会直接调用相

应的IMP方法，这就完成了方法的调用。如果cache落空，则跳转至LCacheMiss标签，调用MethodTableLookup方法，这个方法将IMP的值存在r11寄存器里，之后 `jmp *%r11` 从IMP开始执行，完成方法调用。MethodTableLookup函数实现如下，

```
.macro MethodTableLookup
    MESSENGER_END_SLOW
    SaveRegisters
    // _class_lookupMethodAndLoadCache3(receiver, selector, class)
    movq    $0, %a1
    movq    $1, %a2
    movq    %r11, %a3
    call    __class_lookupMethodAndLoadCache3
    // IMP is now in %rax
    movq    %rax, %r11
    RestoreRegisters
.endmacro
```

可以看到其实际上将receiver（即self)， selector(即_cmd)，class(即self->isa)传递给了_class_lookupMethodAndLoadCache3这个函数，查看该函数的实现后，欢迎重新回到C语言的世界。

```
#objc-class-old.mm#
IMP _class_lookupMethodAndLoadCache3(id obj, SEL sel, Class cls)
{
    return lookUpImpOrForward(cls, sel, obj,
                              YES/*initialize*/, NO/*cache*/, YES/*resolver*/);
}
```

这个函数进一步调用了 `lookUpImpOrForward` ，并把cache标签置为NO，意味着忽略第一次不加锁的cache查找。这个函数的返回值要么是对应方法的IMP指针，要么是一个__objc_msgForward_impcache汇编方法的入口，后者对应着消息转发机制，即如果在该对象及其继承链上方的的对象都找不到选择器_cmd的响应方法的话，就调用消息转发函数尝试将该消息转发给其他对象。下面是lookUpImpOrForward的实现，由于代码过长，注释将写在代码之中。

```
#objc-class-old.mm#
IMP lookUpImpOrForward(Class cls, SEL sel, id inst,
                       bool initialize, bool cache, bool resolver)
{
    Class curClass;                  // 当前类对象
    IMP methodPC = nil;              // 用于保存最终查找到的函数指针并返回
    Method meth;                     // 定义了方法的一个结构体，可通过meth->imp获取函数指针
    bool triedResolver = NO;         // 方法解析的标志变量

    methodListLock.assertUnlocked();
    // 不加锁地查找cache，由于之前cache落空，所以肯定找不到，就忽略
    // Optimistic cache lookup
    if (cache) {
        methodPC = _cache_getImp(cls, sel);
        if (methodPC) return methodPC;
    }
    // Check for freed class
    if (cls == _class_getFreedObjectClass())
        return (IMP) _freedHandler;
    // 确保该类已被初始化，如果没有就调用类方法+initialize，这里也说明了为什么OC的类会在
    // 第一次接收消息后调用+initialize进行初始化，相反的，如果想要代码在类注册runtime的
    // 时候就运行，可以将代码写在+load方法里
```

```
    // Check for +initialize
    if (initialize  &&  !cls->isInitialized()) {
        _class_initialize (_class_getNonMetaClass(cls, inst));
        // If sel == initialize, _class_initialize will send +initialize and
        // then the messenger will send +initialize again after this
        // procedure finishes. Of course, if this is not being called
        // from the messenger then it won't happen. 2778172
    }
    // The lock is held to make method-lookup + cache-fill atomic
    // with respect to method addition. Otherwise, a category could
    // be added but ignored indefinitely because the cache was re-filled
    // with the old value after the cache flush on behalf of the category.
    // 上述英文已述：对消息查找和填充cache加锁，由于填充cache是写操作，所以需要对其
    // 加锁以免加入了category之后的cache被旧的cache冲掉，导致category失效。

    // 实际上，如果cache没有命中，但在方法列表中找到了对应的IMP，函数也是会进行cache
    // 写入操作。
 retry:
    methodListLock.lock();
    // 在开启GC选项后忽略retain, release等方法(猜测GC 是 Garbage Collection)
    // 这也体现了OC的灵活性，runtime完全有权力忽略一些方法
    if (ignoreSelector(sel)) {
        methodPC = _cache_addIgnoredEntry(cls, sel);
        goto done;
    }
    // 在加锁的状态下再查找一次cache，如果命中就直接返回IMP指针
    // 个人认为再次在加锁状态下查找是因为在与上次查找的间隙中可能
    // 有其他类填充了这个cache
    methodPC = _cache_getImp(cls, sel);
    if (methodPC) goto done;

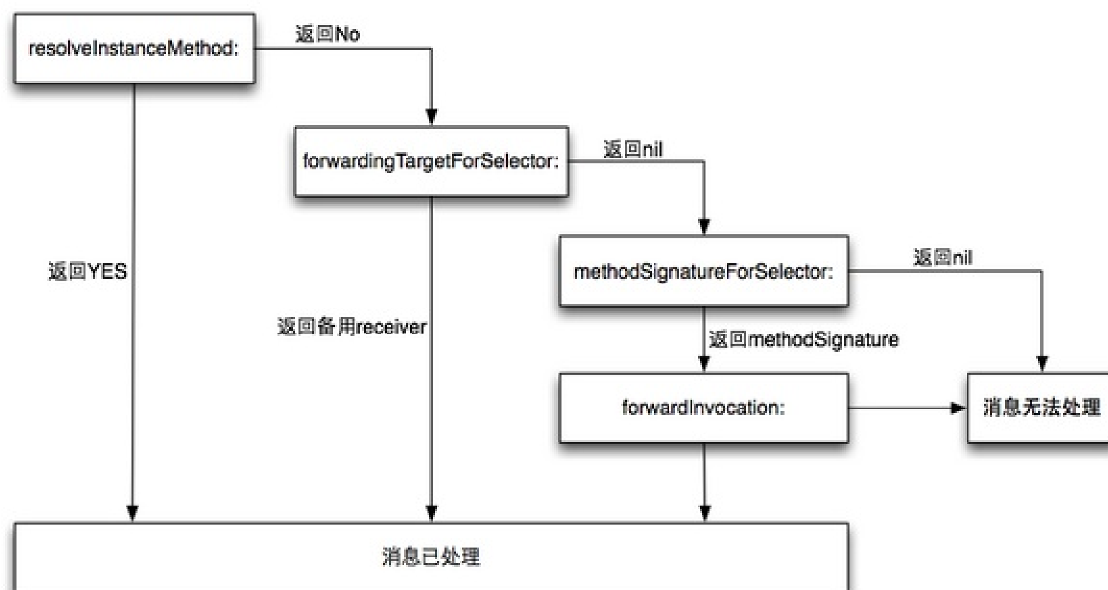
    // 如果还是没有命中的话就查找该类的方法列表
    meth = _class_getMethodNoSuper_nolock(cls, sel);
    if (meth) {
        // 命中，填充cache，返回IMP指针
        log_and_fill_cache(cls, cls, meth, sel);
        methodPC = method_getImplementation(meth);
        goto done;
    }

    // 没有命中，沿着class的继承链向上查找，最后找到的是NSObject(NSProxy除外)
    // 而NSObject的superclass为nil
    curClass = cls;
    while ((curClass = curClass->superclass)) {
        // 尝试从超类的cache中加载
        meth = _cache_getMethod(curClass, sel, _objc_msgForward_impcache);
        if (meth) {
            // 如果不是forward
            if (meth != (Method)1) {
                // 在超类中找到IMP，在当前类中进行cache
                log_and_fill_cache(cls, curClass, meth, sel);
                methodPC = method_getImplementation(meth);
                goto done;
            }
            else {
            // 找到forward，跳出循环
                // Found a forward:: entry in a superclass.
                // Stop searching, but don't cache yet; call method
                // resolver for this class first.
                break;
            }
        }
        // 超类cache没有命中，从超类的方法列表寻找
        meth = _class_getMethodNoSuper_nolock(curClass, sel);
        if (meth) {
            log_and_fill_cache(cls, curClass, meth, sel);
            methodPC = method_getImplementation(meth);
            goto done;
        }
    }
    // 使用方法解析并再尝试一次
    if (resolver  &&  !triedResolver) {
```

```
            methodListLock.unlock();
            _class_resolveMethod(cls, sel, inst);
            triedResolver = YES;
            goto retry;
        }
        // 没有找到IMP指针，方法解析也没有用，使用消息转发，并将其填充入cache
        _cache_addForwardEntry(cls, sel);
        methodPC = _objc_msgForward_impcache;
     done:
        methodListLock.unlock();
        // paranoia: look for ignored selectors with non-ignored implementations
        assert(!(ignoreSelector(sel)  &&  methodPC != (IMP)&_objc_ignored_method));
        return methodPC;
  }
```

我们可以看到每一个类都维护了一个cache，在一个对象调用runtime的objc_msgSend函数后，runtime在接收者所属的类的cache中查找与_cmd所对应的IMP，如果没有命中就寻找当前类的方法列表，再找不到就跳入while循环寻找超类的cache和方法列表，如果这些方法都失效，就调用 `_class_resolveMethod` 查找正在插入这个类的方法，之后再重新尝试整一个流程，如果最后还是没能找到一个对应的IMP，则调用消息转发机制。

动态消息解析



如下用于描述动态消息解析的流程:

- 1.通过 resolveInstanceMethod 得知方法是否为动态添加，YES则通过 class_addMethod 动态添加方法，处理消息，否则进入下一步。dynamic 属性就与这个过程有关，当一个属性声明为 dynamic 时 就是告诉编译器：开发者一定会添加 setter/getter 的实现，而编译时不用自动生成。
- 2.这步会进入 forwardingTargetForSelector 用于指定哪个对象来响应消息。如果返回nil 则进入第三步。这种方式把消息原封不动地转发给目标对象，有着比较高的效率。如果不能自己的类里面找到替代方法，可以重载这个方法，然后把消息转给其他的对象。

- 3.这步调用 methodSignatureForSelector 进行方法签名，这可以将函数的参数类型和返回值封装。如果返回 nil 说明消息无法处理并报错 `unrecognized selector sent to instance`，如果返回 methodSignature，则进入 forwardInvocation ，在这里可以修改实现方法，修改响应对象等，如果方法调用成功，则结束。如果依然不能正确响应消息，则报错 `unrecognized selector sent to instance`.

可以利用 2、3 中的步骤实现对接受消息对象的转移，可以实现“多重继承”的效果。

参考资料

- http://yulingtianxia.com/blog/2014/11/05/objective-c-runtime/
- http://www.cocoawithlove.com/2010/01/what-is-meta-class-in-objective-c.html
- https://github.com/opensource-apple/objc4

推荐首先阅读 内存管理

Objective-C 中的内存分配

在 Objective-C 中，对象通常是使用 `alloc` 方法在堆上创建的。 `[NSObject alloc]` 方法会在对堆上分配一块内存，按照 `NSObject` 的内部结构填充这块儿内存区域。

一旦对象创建完成，就不可能再移动它了。因为很可能有很多指针都指向这个对象，这些指针并没有被追踪。因此没有办法在移动对象的位置之后更新全部的这些指针。

MRC 与 ARC

Objective-C中提供了两种内存管理机制：MRC（MannulReference Counting）和 ARC(Automatic Reference Counting)，分别提供对内存的手动和自动管理，来满足不同的需求。现在苹果推荐使用 ARC 来进行内存管理。

MRC

对象操作的四个类别

对象操作	**OC**中对应的方法	对应的 **retainCount** 变化
生成并持有对象	alloc/new/copy/mutableCopy等	+1
持有对象	retain	+1
释放对象	release	-1
废弃对象	dealloc	-

注意：

- ~~这些对象操作的方法其实并不包括在OC中，而是包含在Cocoa框架下的Foundation框架中。~~从 iOS 7 开始，这些方法被移动到了 Runtime 当中，可以在 objc4-680 NSObject.h 找到。
- 对象的 `reatinCount` 属性并没有实际上的参考价值，参考苹果官方文档《Practical Memory Management》.

四个法则

- 自己生成的对象，自己持有。
- 非自己生成的对象，自己也能持有。
- 不在需要自己持有对象的时候，释放。
- 非自己持有的对象无需释放。

如下是四个黄金法则对应的代码示例：

```
/*
 * 自己生成并持有该对象
 */
 id obj0 = [[NSObeject alloc] init];
 id obj1 = [NSObeject new];
```

```
/*
 * 持有非自己生成的对象
 */
id obj = [NSArray array]; // 非自己生成的对象，且该对象存在，但自己不持有
[obj retain]; // 自己持有对象
```

```
/*
 * 不在需要自己持有的对象的时候，释放
 */
id obj = [[NSObeject alloc] init]; // 此时持有对象
[obj release]; // 释放对象
/*
 * 指向对象的指针仍就被保留在obj这个变量中
 * 但对象已经释放，不可访问
 */
```

```
/*
 * 非自己持有的对象无法释放
 */
id obj = [NSArray array]; // 非自己生成的对象，且该对象存在，但自己不持有
[obj release]; // ~~~此时将运行时crash 或编译器报error~~~ 非 ARC 下，调用该方法会导致编译器报 issues。此操作的行为是未定义的，可能会导致运行时 crash 或者其它未知行为
```

其中 `非自己生成的对象，且该对象存在，但自己不持有` 这个特性是使用 `autorelease` 来实现的，示例代码如下：

```
- (id) getAObjNotRetain {
    id obj = [[NSObject alloc] init]; // 自己持有对象
    [obj autorelease]; // 取得的对象存在，但自己不持有该对象
    return obj;
}
```

`autorelease` 使得对象在超出生命周期后能正确的被释放(通过调用release方法)。在调用 `release` 后，对象会被立即释放，而调用 `autorelease` 后，对象不会被立即释放，而是注册到 `autoreleasepool` 中，经过一段时间后 `pool` 结束，此时调用release方法，对象被释放。

在MRC的内存管理模式下，与对变量的管理相关的方法有：retain, release 和 autorelease。retain 和 release 方法操作的是引用记数，当引用记数为零时，便自动释放内存。并且可以用 NSAutoreleasePool 对象，对加入自动释放池（autorelease 调用）的变量进行管理，当 drain 时回收内存。

ARC

ARC 是苹果引入的一种自动内存管理机制，会根据引用计数自动监视对象的生存周期，实现方式是在编译时期自动在已有代码中插入合适的内存管理代码以及在 Runtime 做一些优化。

变量标识符

在ARC中与内存管理有关的变量标识符，有下面几种：

- `__strong`
- `__weak`
- `__unsafe_unretained`
- `__autoreleasing`

`__strong` 是默认使用的标识符。只有还有一个强指针指向某个对象，这个对象就会一直存活。

`__weak` 声明这个引用不会保持被引用对象的存活，如果对象没有强引用了，弱引用会被置为 nil

`__unsafe_unretained` 声明这个引用不会保持被引用对象的存活，如果对象没有强引用了，它不会被置为 nil。如果它引用的对象被回收掉了，该指针就变成了野指针。

`__autoreleasing` 用于标示使用引用传值的参数（id *），在函数返回时会被自动释放掉。

变量标识符的用法如下：

```
Number* __strong num = [[Number alloc] init];
```

注意 `__strong` 的位置应该放到 `*` 和变量名中间，放到其他的位置严格意义上说是不正确的，只不过编译器不会报错。

属性标识符

类中的属性也可以加上标志符：

```
@property (assign/retain/strong/weak/unsafe_unretained/copy) Number* num
```

`assign` 表明 setter 仅仅是一个简单的赋值操作，通常用于基本的数值类型，例如 `CGFloat` 和 `NSInteger` 。

`strong` 表明属性定义一个拥有者关系。当给属性设定一个新值的时候，首先这个值进行 `retain` ，旧值进行 `release` ，然后进行赋值操作。

`weak` 表明属性定义了一个非拥有者关系。当给属性设定一个新值的时候，这个值不会进行 `retain`，旧值也不会进行 `release`， 而是进行类似 `assign` 的操作。不过当属性指向的对象被销毁时，该属性会被置为nil。

`unsafe_unretained` 的语义和 `assign` 类似，不过是用于对象类型的，表示一个非拥有(unretained)的，同时也不会在对象被销毁时置为nil的(unsafe)关系。

`copy` 类似于 `strong`，不过在赋值时进行 `copy` 操作而不是 `retain` 操作。通常在需要保留某个不可变对象（NSString最常见），并且防止它被意外改变时使用。

错误使用属性标识符的后果

如果我们给一个原始类型设置 `strong\weak\copy`，编译器会直接报错：

> Property with 'retain (or strong)' attribute must be of object type

设置为 `unsafe_unretained` 倒是可以通过编译，只是用起来跟 `assign` 也没有什么区别。

反过来，我们给一个 NSObject 属性设置为 assign，编译器会报警：

> Assigning retained object to unsafe property; object will be released after assignment

正如警告所说的，对象在赋值之后被立即释放，对应的属性也就成了野指针，运行时跑到属性有关操作会直接崩溃掉。和设置成 `unsafe_unretained` 是一样的效果（设置成 `weak` 不会崩溃）。

`unsafe_unretained` 的用处

`unsafe_unretained` 差不多是实际使用最少的一个标识符了，在使用中它的用处主要有下面几点：

1. 兼容性考虑。iOS4 以及之前还没有引入 `weak`，这种情况想表达弱引用的语义只能使用 `unsafe_unretained`。这种情况现在已经很少见了。
2. 性能考虑。使用 `weak` 对性能有一些影响，因此对性能要求高的地方可以考虑使用 `unsafe_unretained` 替换 `weak`。一个例子是 YYModel 的实现，为了追求更高的性能，其中大量使用 `unsafe_unretained` 作为变量标识符。

引用循环

当两个对象互相持有对方的强引用，并且这两个对象的引用计数都不是0的时候，便造成了引用循环。

要想破除引用循环，可以从以下几点入手：

- 注意变量作用域，使用 `autorelease` 让编译器来处理引用
- 使用弱引用(weak)
- 当实例变量完成工作后，将其置为nil

Autorelease Pool

Autorelase Pool 提供了一种可以允许你向一个对象延迟发送 `release` 消息的机制。当你想放弃一个对象的所有权，同时又不希望这个对象立即被释放掉（例如在一个方法中返回一个对象时），Autorelease Pool 的作用就显现出来了。

所谓的延迟发送 `release` 消息指的是，当我们把一个对象标记为 `autorelease` 时:

```
NSString* str = [[[NSString alloc] initWithString:@"hello"] autorelease];
```

这个对象的 retainCount 会+1，但是并不会发生 release。当这段语句所处的 autoreleasepool 进行 drain 操作时，所有标记了 `autorelease` 的对象的 retainCount 会被 -1。即 `release` 消息的发送被延迟到 pool 释放的时候了。

在 ARC 环境下，苹果引入了 `@autoreleasepool` 语法，不再需要手动调用 `autorelease` 和 `drain` 等方法。

Autorelease Pool 的用处

在 ARC 下，我们并不需要手动调用 autorelease 有关的方法，甚至可以完全不知道 autorelease 的存在，就可以正确管理好内存。因为 Cocoa Touch 的 Runloop 中，每个 runloop circle 中系统都自动加入了 Autorelease Pool 的创建和释放。

当我们需要创建和销毁大量的对象时，使用手动创建的 autoreleasepool 可以有效的避免内存峰值的出现。因为如果不手动创建的话，外层系统创建的 pool 会在整个 runloop circle 结束之后才进行 drain，手动创建的话，会在 block 结束之后就进行 drain 操作。详情请参考苹果官方文档。一个普遍被使用的例子如下：

```
for (int i = 0; i < 100000000; i++)
{
    @autoreleasepool
    {
        NSString* string = @"ab c";
        NSArray* array = [string componentsSeparatedByString:string];
    }
}
```

如果不使用 autoreleasepool ，需要在循环结束之后释放 100000000 个字符串，如果 使用的话，则会在每次循环结束的时候都进行 release 操作。

Autorelease Pool 进行 Drain 的时机

如上面所说，系统在 runloop 中创建的 autoreleaspool 会在 runloop 一个 event 结束时进行释放操作。我们手动创建的 autoreleasepool 会在 block 执行完成之后进行 drain 操作。需要注意的是：

- 当 block 以异常（exception）结束时，pool 不会被 drain
- Pool 的 drain 操作会把所有标记为 autorelease 的对象的引用计数减一，但是并不意味着这个对象一定会被释放掉，我们可以在 autorelease pool 中手动 retain 对象，以延长它

的生命周期（在 MRC 中）。

main.m 中 Autorelease Pool 的解释

大家都知道在 iOS 程序的 main.m 文件中有类似这样的语句：

```
int main(int argc, char * argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class
]));
    }
}
```

在面试中问到有关 autorelease pool 有关的知识也多半会问一下，这里的 pool 有什么作用，能不能去掉之类。在这里我们分析一下。

根据苹果官方文档， UIApplicationMain 函数是整个 app 的入口，用来创建 application 对象（单例）和 application delegate。尽管这个函数有返回值，但是实际上却永远不会返回，当按下 Home 键时，app 只是被切换到了后台状态。

同时参考苹果关于 Lifecycle 的官方文档，UIApplication 自己会创建一个 main run loop，我们大致可以得到下面的结论：

1. main.m 中的 UIApplicationMain 永远不会返回，只有在系统 kill 掉整个 app 时，系统会把应用占用的内存全部释放出来。
2. 因为(1)， UIApplicationMain 永远不会返回，这里的 autorelease pool 也就永远不会进入到释放那个阶段
3. 在 (2) 的基础上，假设有些变量真的进入了 main.m 里面这个 pool（没有被更内层的 pool 捕获），那么这些变量实际上就是被泄露的。这个 autorelease pool 等于是把这种泄露情况给隐藏起来了。
4. UIApplication 自己会创建 main run loop，在 Cocoa 的 runloop 中实际上也是自动包含 autorelease pool 的，因此 main.m 当中的 pool 可以认为是没有必要的。

在基于 AppKit 框架的 Mac OS 开发中， main.m 当中就是不存在 autorelease pool 的，也进一步验证了我们得到的结论。不过因为我们看不到更底层的代码，加上苹果的文档中不建议修改 main.m ，所以我们也没有理由就直接把它删掉（亲测，删掉之后不影响 App 运行，用 Instruments 也看不到泄露）。

Autorelease Pool 与函数返回值

如果一个函数的返回值是指向一个对象的指针，那么这个对象肯定不能在函数返回之前进行 release，这样调用者在调用这个函数时得到的就是野指针了，在函数返回之后也不能立刻就 release，因为我们不知道调用者是不是 retain 了这个对象，如果我们直接 release 了，可能导致后面在使用这个对象时它已经成为 nil 了。

为了解决这个纠结的问题， Objective-C 中对对象指针的返回值进行了区分，一种叫做 *retained return value*，另一种叫做 *unretained return value*。前者表示调用者拥有这个返回值，后者表示调用者不拥有这个返回值，按照“谁拥有谁释放”的原则，对于前者调用者是要负责释放的，对于后者就不需要了。

按照苹果的命名 convention，以 `alloc`, `copy`, `init`, `mutableCopy` 和 `new` 这些方法打头的方法，返回的都是 retained return value，例如 `[[NSString alloc] initWithFormat:]`，而其他的则是 unretained return value，例如 `[NSString stringWithFormat:]`。我们在编写代码时也应该遵守这个 convention。

我们分别在 MRC 和 ARC 情况下，分析一下两种返回值类型的区别。

MRC

在 MRC 中我们需要关注这两种函数返回类型的区别，否则可能会导致内存泄露。

对于 *retained return value*，需要负责释放

假设我们有一个 property 定义如下：

```
@property (nonatomic, retain) NSObject *property;
```

在对其赋值的时候，我们应该使用：

```
self.property = [[[NSObject alloc] init] autorelease];
```

然后在 dealloc 方法中加入：

```
[_property release];
_property = nil;
```

这样内存的情况大体是这样的：

1. init 把 retain count 增加到 1
2. 赋值给 self.property ，把 retain count 增加到 2
3. 当 runloop circle 结束时，autorelease pool 执行 drain，把 retain count 减为 1
4. 当整个对象执行 dealloc 时， release 把 retain count 减为 0，对象被释放

可以看到没有内存泄露发生。

如果我们只是使用：

```
self.property = [[NSObject alloc] init];
```

这一条语句会导致 retain count 增加到 2，而我们少执行了一次 release，就会导致 retain count 不能被减为 0 。

另外，我们也可以使用临时变量：

```
NSObject * a = [[NSObject alloc] init];
self.property = a;
[a release];
```

这种情况，因为对 a 执行了一次 release，所有不会出现上面那种 retain count 不能减为 0 的情况。

注意：现在大家基本都是 ARC 写的比较多，会忽略这一点，但是根据上面的内容，我们看到在 MRC 中直接对 self.proprety 赋值和先赋给临时变量，再赋值给 self.property，确实是有区别的！我在面试中就被问到这一点了。

我们在编写自己的代码时，也应该遵守上面的原则，同样是使用 autorelease：

```
// 注意函数名的区别
+ (MyCustomClass *) myCustomClass
{
    return [[[MyCustomClass alloc] init] autorelease]; // 需要 autorelease
}
- (MyCustomClass *) initWithName:(NSString *) name
{
    return [[MyCustomClass alloc] init]; // 不需要 autorelease
}
```

对于 *unretained return value*，不需要负责释放

当我们调用非 alloc，init 系的方法来初始化对象时（通常是工厂方法），我们不需要负责变量的释放，可以当成普通的临时变量来使用：

```
NSString *name = [NSString stringWithFormat:@"%@ %@", firstName, lastName];
self.name = name
// 不需要执行 [name release]
```

ARC

在 ARC 中我们完全不需要考虑这两种返回值类型的区别，ARC 会自动加入必要的代码，因此我们可以放心大胆地写：

```
self.property = [[NSObject alloc] init];
self.name = [NSString stringWithFormat:@"%@ %@", firstName, lastName];
```

以及在自己写的函数中：

```
+ (MyCustomClass *) myCustomClass
{
    return [[MyCustomClass alloc] init]; // 不用 autorelease
}
```

这些写法都是 OK 的，也不会出现内存问题。

为了进一步理解 ARC 是如何做到这一点的，我们可以参考 Clang 的文档。

对于 retained return value， Clang 是这样做的：

> When returning from such a function or method, ARC retains the value at the point of evaluation of the return statement, before leaving all local scopes.
>
> When receiving a return result from such a function or method, ARC releases the value at the end of the full-expression it is contained within, subject to the usual optimizations for local values.

可以看到基本上 ARC 就是帮我们在代码块结束的时候进行了 release：

```
NSObject * a = [[NSObject alloc] init];
self.property = a;
//[a release]; 我们不需要写这一句，因为 ARC 会帮我们把这一句加上
```

对于 unretained return value：

> When returning from such a function or method, ARC retains the value at the point of evaluation of the return statement, then leaves all local scopes, and then balances out the retain while ensuring that the value lives across the call boundary. In the worst case, this may involve an autorelease, but callers must not assume that the value is actually in the autorelease pool.
>
> ARC performs no extra mandatory work on the caller side, although it may elect to do something to shorten the lifetime of the returned value.

这个和我们之前在 MRC 中做的不是完全一样。ARC 会把对象的生命周期延长，确保调用者能拿到并且使用这个返回值，但是并不一定会使用 autorelease，文档写的是在 worst case 的情况下才可能会使用，因此调用者不能假设返回值真的就在 autorelease pool 中。从性能的角度，这种做法也是可以理解的。如果我们能够知道一个对象的生命周期最长应该有多长，也就没有必要使用 autorelease 了，直接使用 release 就可以。如果很多对象都使用 autorelease 的话，也会导致整个 pool 在 drain 的时候性能下降。

weak 与 autorelease

众所周知，weak 不会持有对象，当给一个 weak 赋以一个自己生成的对象（即上面提到的 retained return value）后，对象会立马被释放。

一个很常见的 warning 就是 Assigning retained object to weak variable, object will be released after assignment.

但是我们前面也提到了，可以持有非自己生成的对象，这通过 autorelease 实现。

那么如果一个 weak 被赋以一个非自己生成的对象（即上面提到的 unretained return value）呢？代码如下：

```
NSNumber __weak *number = [NSNumber numberWithInt:100];
NSLog(@"number = %@", number);
```

这种情况下是可以正确打印值的。

clang的文档 是这么说的：这种情况下，weak 并不会立即释放，而是会通过 `objc_loadWeak` 这个方法注册到 AutoreleasePool 中，以延长生命周期。

ARC 下是否还有必要在 dealloc 中把属性置为 nil？

为了解决这个问题，首先让我们理清楚属性是个什么存在。属性(property) 实际上就是一种语法糖，每个属性背后都有实例变量(Ivar)做支持，编译器会帮我们自动生成有关的 setter 和 getter，对于下面的 property：

```
@interface Counter : NSObject
@property (nonatomic, retain) NSNumber *count;
@end;
```

生成的 getter 和 setter 类似下面这样：

```
- (NSNumber *)count {
    return _count;
}
- (void)setCount:(NSNumber *)newCount {
    [newCount retain];
    [_count release];
    // Make the new assignment.
    _count = newCount;
}
```

Property 这部分对于 MRC 和 ARC 都是适用的。

有了这部分基础，我们再来理解一下把属性置为 nil 这个步骤。首先要明确一点，在 MRC 下，我们并不是真的把属性置为 nil，而是把 Ivar 置为 nil。

```
[_property release];
_property = nil;
```

如果用 self.property 的话还会调用 setter，里面可能存在某些不应该在 dealloc 时运行的代码。

对于 ARC 来说，系统会自动在 dealloc 的时候把所有的 Ivar 都执行 release，因此我们也就没有必要在 dealloc 中写有关 release 的代码了。

在 **ARC** 下把变量置为 **nil** 有什么效果？什么情况下需要把变量置为 **nil**？

在上面有关 property 的内容基础上，我们知道用:

```
self.property = nil
```

实际上就是手动执行了一次 release。而对于临时变量来说：

```
NSObject *object = [[NSObject alloc] init];
object = nil;
```

置为 nil 这一句其实没什么用（除了让 object 在下面的代码里不能再使用之外），因为上面我们讨论过 ，ARC 下的临时变量是受到 Autorelease Pool 的管理的，会自动释放。

因为 ARC 下我们不能再使用 release 函数，把变量置为 nil 就成为了一种释放变量的方法。真正需要我们把变量置为 nil 的，通常就是在使用 block 时，用于破除循环引用：

```
MyViewController * __block myController = [[MyViewController alloc] init…];
// ...
myController.completionHandler =  ^(NSInteger result) {
    [myController dismissViewControllerAnimated:YES completion:nil];
    myController = nil;
};
```

在 YTKNetwork 这个项目中，也可以看到类似的代码：

```
- (void)clearCompletionBlock {
    // nil out to break the retain cycle.
    self.successCompletionBlock = nil;
    self.failureCompletionBlock = nil;
}
```

ARC 在运行时期的优化

上面提到对于 unretained return value， ARC “并不一定会使用 autorelease”，下面具体解释一下。

ARC 所做的事情并不仅仅局限于在编译期找到合适的位置帮你插入合适的 `release` 等等这样的内存管理方法，其在运行时期也做了一些优化，如下是两个优化的例子：

1. 合并对称的引用计数操作。比如将 +1/-1/+1/-1 直接置为 0.
2. 巧妙地跳过某些情况下 `autorelease` 机制的调用。

其中第二个优化，是 ARC 针对 `autorelease` 返回值提供的一套优化策略，大体的流程如下：

当方法全部基于 ARC 实现时，在方法 return 的时候，ARC 会调用 `objc_autoreleaseReturnValue()` 以替代 MRC 下的 `autorelease`。在 MRC 下需要 retain 的位置，ARC 会调用 `objc_retainAutoreleasedReturnValue()`。因此下面的 ARC 代码：

```
+ (instancetype)createSark {
    return [self new];
}
// caller
Sark *sark = [Sark createSark];
```

实际上会被改写成类似这样：

```
+ (instancetype)createSark {
    id tmp = [self new];
    return objc_autoreleaseReturnValue(tmp); // 代替我们调用autorelease
}
// caller
id tmp = objc_retainAutoreleasedReturnValue([Sark createSark]) // 代替我们调用retain
Sark *sark = tmp;
objc_storeStrong(&sark, nil); // 相当于代替我们调用了release
```

有了这个基础，ARC 可以使用一些优化技术。在调用 `objc_autoreleaseReturnValue()` 时，会在栈上查询 return address 以确定 return value 是否会被直接传给 `objc_retainAutoreleasedReturnValue()`。 如果没传，说明返回值不能直接从提供方发送给接收方，这时就会调用 `autorelease`。反之，如果返回值能顺利的从提供方传送给接收方，那么就会直接跳过 `autorelease` 过程，并且修改 return address 以跳过 `objc_retainAutoreleasedReturnValue()` 过程，这样就跳过了整个 `autorelease` 和 `retain` 的过程。

> 核心思想：当返回值被返回之后，紧接着就需要被 retain 的时候，没有必要进行 autorelease + retain，直接什么都不要做就好了。

另外，当函数的调用方是非 ARC 环境时，ARC 还会进行更多的判断，在这里不再详述，详见《黑幕背后的 Autorelease》。

关于如何写一个检测循环引用的工具

Instrument 为我们提供了 Allocations/Leaks 这样好用的工具用来检测 memory leak 的工具。如下是内存泄露的两种类型：

- Leaked memory: Memory unreferenced by your application that cannot be used again or freed (also detectable by using the Leaks instrument).
- Abandoned memory: Memory still referenced by your application that has no useful purpose.

其中 Leaks 工具主要用来检测 Leaked memory，在 MRC 时代 程序员会经常忘记写 release 方法导致内存泄露，在 ARC 时代这种已经不太常见。(ARC时代 主要的Leaked Memory 来自于底层 C 语言以及 一些由 C 写成的底层库，往往会因为忘记手工 free 而导致 leak)。

Allocations 工具主要用来检测 Abandoned memory. 主要思路是在一个时间切片内检测对象的声明周期以观察内存是否会无限增长。通过 hook 掉 alloc，dealloc，retain，release 等方法，来记录对象的生命周期。

参考资料

- Objective-C内存管理MRC与ARC
- 10个Objective-C基础面试题，iOS面试必备
- 黑幕背后的 Autorelease
- Objective-C Autorelease Pool 的实现原理
- How does objc_retainAutoreleasedReturnValue work?
- https://stackoverflow.com/questions/9784762/strong-weak-retain-unsafe-unretained-assign
- https://stackoverflow.com/questions/29350634/ios-autoreleasepool-in-main-and-arc-alloc-release
- https://stackoverflow.com/questions/6588211/why-do-the-ios-main-m-templates-include-a-return-statement-and-an-autorelease-po
- https://stackoverflow.com/questions/2702548/if-the-uiapplicationmain-never-returns-then-when-does-the-autorelease-pool-get
- https://stackoverflow.com/questions/6055274/use-autorelease-when-setting-a-retain-property-using-dot-syntax
- https://stackoverflow.com/questions/17601274/arc-and-autorelease
- https://stackoverflow.com/questions/8292060/arc-equivalent-of-autorelease
- https://stackoverflow.com/questions/7906804/do-i-set-properties-to-nil-in-dealloc-when-using-arc
- http://wereadteam.github.io/2016/02/22/MLeaksFinder/?from=singlemessage&isappinstalled=0
- http://clang.llvm.org/docs/AutomaticReferenceCounting.html#arc-runtime-objc-loadweak

Runloop

Runloop 是和线程紧密相关的一个基础组件，是很多线程有关功能的幕后功臣。尽管在平常使用中几乎不太会直接用到，理解 Runloop 有利于我们更加深入地理解 iOS 的多线程模型。

Runloop 基本概念

Runloop 是什么？Runloop 还是比较顾名思义的一个东西，说白了就是一种循环，只不过它这种循环比较高级。一般的 while 循环会导致 CPU 进入忙等待状态，而 Runloop 则是一种“闲”等待，这部分可以类比 Linux 下的 epoll。当没有事件时，Runloop 会进入休眠状态，有事件发生时， Runloop 会去找对应的 Handler 处理事件。Runloop 可以让线程在需要做事的时候忙起来，不需要的话就让线程休眠。

盗一张苹果官方文档的图，也是几乎每个讲 Runloop 的文章都会引用的图，大体说明了 Runloop 的工作模式：



图中展现了 Runloop 在线程中的作用：从 input source 和 timer source 接受事件，然后在线程中处理事件。

Runloop 与线程

Runloop 和线程是绑定在一起的。每个线程（包括主线程）都有一个对应的 Runloop 对象。我们并不能自己创建 Runloop 对象，但是可以获取到系统提供的 Runloop 对象。

主线程的 Runloop 会在应用启动的时候完成启动，其他线程的 Runloop 默认并不会启动，需要我们手动启动。

Input Source 和 Timer Source

这两个都是 Runloop 事件的来源，其中 Input Source 又可以分为三类

- Port-Based Sources，系统底层的 Port 事件，例如 CFSocketRef ，在应用层基本用不到
- Custom Input Sources，用户手动创建的 Source
- Cocoa Perform Selector Sources， Cocoa 提供的 performSelector 系列方法，也是一种事件源

Timer Source 顾名思义就是指定时器事件了。

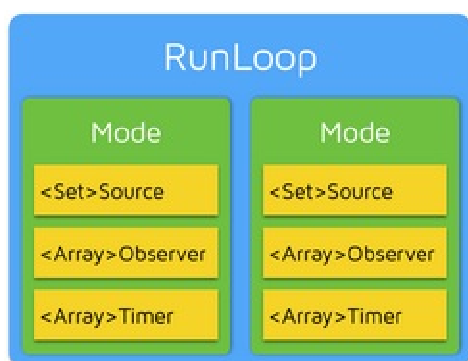
Runloop Observer

Runloop 通过监控 Source 来决定有没有任务要做，除此之外，我们还可以用 Runloop Observer 来监控 Runloop 本身的状态。 Runloop Observer 可以监控下面的 runloop 事件：

- The entrance to the run loop.
- When the run loop is about to process a timer.
- When the run loop is about to process an input source.
- When the run loop is about to go to sleep.
- When the run loop has woken up, but before it has processed the event that woke it up.
- The exit from the run loop.

Runloop Mode

在监视与被监视中，Runloop 要处理的事情还挺复杂的。为了让 Runloop 能专心处理自己关心的那部分事情，引入了 Runloop Mode 概念。



如图所示，Runloop Mode 实际上是 Source，Timer 和 Observer 的集合，不同的 Mode 把不同组的 Source，Timer 和 Observer 隔绝开来。Runloop 在某个时刻只能跑在一个 Mode 下，处理这一个 Mode 当中的 Source，Timer 和 Observer。

苹果文档中提到的 Mode 有五个，分别是：

- NSDefaultRunLoopMode
- NSConnectionReplyMode
- NSModalPanelRunLoopMode
- NSEventTrackingRunLoopMode
- NSRunLoopCommonModes

iOS 中公开暴露出来的只有 NSDefaultRunLoopMode 和 NSRunLoopCommonModes。 NSRunLoopCommonModes 实际上是一个 Mode 的集合，默认包括 NSDefaultRunLoopMode 和 NSEventTrackingRunLoopMode。

与 Runloop 相关的坑

日常开发中，与 runLoop 接触得最近可能就是通过 NSTimer 了。一个 Timer 一次只能加入到一个 RunLoop 中。我们日常使用的时候，通常就是加入到当前的 runLoop 的 default mode 中，而 ScrollView 在用户滑动时，主线程 RunLoop 会转到 UITrackingRunLoopMode 。而这个时候， Timer 就不会运行。

有如下两种解决方案：

- 第一种: 设置RunLoop Mode，例如NSTimer,我们指定它运行于 NSRunLoopCommonModes ，这是一个Mode的集合。注册到这个 Mode 下后，无论当前 runLoop 运行哪个 mode ，事件都能得到执行。
- 第二种: 另一种解决Timer的方法是，我们在另外一个线程执行和处理 Timer 事件，然后在主线程更新UI。

在 AFNetworking 3.0 中，就有相关的代码，如下：

```
- (void)startActivationDelayTimer {
    self.activationDelayTimer = [NSTimer
                                 timerWithTimeInterval:self.activationDelay target:self
 selector:@selector(activationDelayTimerFired) userInfo:nil repeats:NO];
    [[NSRunLoop mainRunLoop] addTimer:self.activationDelayTimer forMode:NSRunLoopCommo
nModes];
}
```

这里就是添加了一个计时器，由于指定了 NSRunLoopCommonModes，所以不管 RunLoop 出于什么状态，都执行这个计时器任务。

参考资料

- https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Multithreading/RunLoopManagement/RunLoopManagement.html#//apple_ref/doc/uid/10000057i-CH16-SW1
- http://chun.tips/blog/2014/10/20/zou-jin-run-loopde-shi-jie-%5B%3F%5D-:shi-yao-shi-run-loop%3F/
- http://www.hrchen.com/2013/07/tricky-runloop-on-ios/
- http://www.cocoachina.com/ios/20150601/11970.html
- http://www.cocoachina.com/ios/20111111/3487.html
- http://mobile.51cto.com/iphone-386596.htm
- http://blog.ibireme.com/2015/05/18/runloop/

事件分类

对于 iOS 设备用户来说，他们操作设备的方式主要有三种：触摸屏幕、晃动设备、通过遥控设施控制设备。对应的事件类型有以下三种：

1. 触屏事件（Touch Event）
2. 运动事件（Motion Event）
3. 远端控制事件（Remote-Control Event）

响应者链

当发生事件响应时，必须知道由谁来响应事件。在 iOS 中，由响应者链来对事件进行响应。

所有事件响应的类都是 UIResponder 的子类，响应者链是一个由不同对象组成的层次结构，其中的每个对象将依次获得响应事件消息的机会。当发生事件时，事件首先被发送给第一响应者，第一响应者往往是事件发生的视图，也就是用户触摸屏幕的地方。事件将沿着响应者链一直向下传递，直到被接受并做出处理。一般来说，第一响应者是个视图对象或者其子类对象，当其被触摸后事件被交由它处理，如果它不处理，事件就会被传递给它的视图控制器对象 ViewController（如果存在），然后是它的父视图（superview）对象（如果存在），以此类推，直到顶层视图。接下来会沿着顶层视图（top view）到窗口（UIWindow 对象）再到程序（UIApplication 对象）。如果整个过程都没有响应这个事件，该事件就被丢弃。一般情况下，在响应者链中只要由对象处理事件，事件就停止传递。

一个典型的事件响应路线如下：

```
First Responser --> The Window --> The Application --> nil（丢弃）
```

我们可以通过 `[responder nextResponder]` 找到当前 responder 的下一个 responder，持续这个过程到最后会找到 UIApplication 对象。

通常情况下，我们在 First Responder （一般也就是用户当前触控的 View ）这里就会响应请求，进入下面的事件分发机制。

事件分发

第一响应者（First responder）指的是当前接受触摸的响应者对象（通常是一个 UIView 对象），即表示当前该对象正在与用户交互，它是响应者链的开端。响应者链和事件分发的使命都是找出第一响应者。

iOS 系统检测到手指触摸 (Touch) 操作时会将其打包成一个 UIEvent 对象，并放入当前活动 Application 的事件队列，单例的 UIApplication 会从事件队列中取出触摸事件并传递给单例的 UIWindow 来处理，UIWindow 对象首先会使用 `hitTest:withEvent:` 方法寻找此次 Touch 操

作初始点所在的视图(View)，即需要将触摸事件传递给其处理的视图，这个过程称之为 hit-test view。

`hitTest:withEvent:` 方法的处理流程如下:

- 首先调用当前视图的 `pointInside:withEvent:` 方法判断触摸点是否在当前视图内；
- 若返回 NO, 则 `hitTest:withEvent:` 返回 nil，若返回 YES, 则向当前视图的所有子视图 (subviews) 发送 `hitTest:withEvent:` 消息，所有子视图的遍历顺序是从最顶层视图一直到到最底层视图，即从 subviews 数组的末尾向前遍历，直到有子视图返回非空对象或者全部子视图遍历完毕；
- 若第一次有子视图返回非空对象，则 `hitTest:withEvent:` 方法返回此对象，处理结束；
- 如所有子视图都返回空，则 hitTest:withEvent: 方法返回自身 (self)。

一个示例性的代码实现如下：

```
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event{
    UIView *touchView = self;
    if ([self pointInside:point withEvent:event] &&
       (!self.hidden) &&
       self.userInteractionEnabled &&
       (self.alpha >= 0.01f)) {

        for (UIView *subView in self.subviews) {
            [subview convertPoint:point fromView:self];
            UIView *subTouchView = [subView hitTest:subPoint withEvent:event];
            if (subTouchView) {
                touchView = subTouchView;
                break;
            }
        }
    }else{
        touchView = nil;
    }

    return touchView;
}
```

说明

1. 如果最终 hit-test 没有找到第一响应者，或者第一响应者没有处理该事件，则该事件会沿着响应者链向上回溯，如果 UIWindow 实例和 UIApplication 实例都不能处理该事件，则该事件会被丢弃（这个过程即上面提到的响应值链）；
2. `hitTest:withEvent:` 方法将会忽略隐藏 (hidden=YES) 的视图，禁止用户操作 (`userInteractionEnabled=NO`) 的视图，以及 alpha 级别小于 0.01(alpha<0.01)的视图。如果一个子视图的区域超过父视图的 bound 区域(父视图的 clipsToBounds 属性为 NO，这样超过父视图 bound 区域的子视图内容也会显示)，那么正常情况下对子视图在父视图之外区域的触摸操作不会被识别, 因为父视图的 `pointInside:withEvent:` 方法会返回 NO, 这样就不会继续向下遍历子视图了。当然，也可以重写 `pointInside:withEvent:` 方法来处理这种情况。
3. 我们可以重写 `hitTest:withEvent:` 来达到某些特定的目的。

CYLTabBarController是一个支持自定义 Tab 控件的开源项目。在 TabBar 当中，为了支持 TabBar 按钮大小超过 TabBar Frame 范围时也可以响应，它的实现就是重载了 hitTest 方法：

```
/*
 *
 Capturing touches on a subview outside the frame of its superview
 *
 */
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event
{
    if (!self.clipsToBounds && !self.hidden && self.alpha > 0) {
        for (UIView *subview in self.subviews.reverseObjectEnumerator) {
            CGPoint subPoint = [subview convertPoint:point fromView:self];
            UIView *result = [subview hitTest:subPoint withEvent:event];
            if (result != nil) {
                return result;
            }
        }
    }
    return nil;
}
```

可以看到和上面的示例代码的差距，主要就在于取消了 `pointInside` 函数的检测，让我们可以捕获到当前 Frame 范围以外的子 View 的触控事件。

参考资料

1. CocoaTouch 事件处理流程
2. http://blog.sina.com.cn/s/blog_59fb90df0101ab26.html

UIApplication 的核心作用是提供了 iOS 程序运行期间的控制和协作工作。

每一个程序在运行期必须有且仅有一个 UIApplication（或则其子类）的一个实例。在程序开始运行的时候，UIApplicationMain 函数是程序进入点，这个函数做了很多工作，其中一个重要的工作就是创建一个 UIApplication 的单例实例。在你的代码中你，你可以通过调用 [UIApplication sharedApplication] 来得到这个单例实例的指针。

UIApplication 的一个主要工作是处理用户事件，它会起一个队列，把所有用户事件都放入队列，逐个处理，在处理的时候，它会发送当前事件 到一个合适的处理事件的目标控件。此外，UIApplication 实例还维护一个在本应用中打开的 window 列表（UIWindow 实例），这样它就 可以接触应用中的任何一个 UIView 对象。UIApplication 实例会被赋予一个代理对象，以处理应用程序的生命周期事件（比如程序启动和关闭）、系统事件（比如来电、记事项警告）等等。

UIApplicaion 生命周期

一个 UIApplication 可以有如下几种状态：

- `Not running（未运行）` 程序没启动
- `Inactive（未激活）` 程序在前台运行，不过没有接收到事件。在没有事件处理情况下程序通常停留在这个状态
- `Active（激活）` 程序在前台运行而且接收到了事件。这也是前台的一个正常的模式
- `Background（后台）` 程序在后台而且能执行代码，大多数程序进入这个状态后会在在这个状态上停留一会。时间到之后会进入挂起状态 (Suspended)。有的程序经过特殊的请求后可以长期处于 Background 状态
- `Suspended（挂起）` 程序在后台不能执行代码。系统会自动把程序变成这个状态而且不会发出通知。当挂起时，程序还是停留在内存中的，当系统内存低时，系统就把挂起的程序清除掉，为前台程序提供更多的内存。

常见的代理方法有

1. `(void)applicationWillResignActive:(UIApplication *)application`

   说明：当应用程序将要入非活动状态执行，在此期间，应用程序不接收消息或事件，比如来电话了

2. `(void)applicationDidBecomeActive:(UIApplication *)application`

   说明：当应用程序入活动状态执行，这个刚好跟上面那个方法相反

3. `(void)applicationDidEnterBackground:(UIApplication *)application`

   说明：当程序被推送到后台的时候调用。所以要设置后台继续运行，则在这个函数里面设置即可

4. `(void)applicationWillEnterForeground:(UIApplication *)application`

   说明：当程序从后台将要重新回到前台时候调用，这个刚好跟上面的那个方法相反。

5. `(void)applicationWillTerminate:(UIApplication *)application`

   说明：当程序将要退出是被调用，通常是用来保存数据和一些退出前的清理工作。这个需要设置 UIApplicationExitsOnSuspend 的键值。

6. `(void)applicationDidReceiveMemoryWarning:(UIApplication *)application`

   说明：iPhone 设备只有有限的内存，如果为应用程序分配了太多内存操作系统会终止应用程序的运行，在终止前会执行这个方法，通常可以在这里进行内存清理工作防止程序被终止

7. `(void)applicationSignificantTimeChange:(UIApplication*)application`

   说明：当系统时间发生改变时执行

8. `(void)applicationDidFinishLaunching:(UIApplication*)application`

   说明：当程序载入后执行

下面是一个用于展示整个 App 生命周期的示意图：

UIApplication Delegate Messaging
with Multitasking
V1.2.1

Start Here
App Starts
application:didFinishLaunchingWithOptions:
replaced previous didFinishLaunching
URL
Yes
No
application:handleOpenURL:
applicationDidBecomeActive:
No
Yes
URL
Home Button or Task Switcher
Foreground Run Loop
Interruption, e.g. Phone Call
applicationWillResignActive:
applicationWillResignActive:
applicationDidEnterBackground:
Yes
User picks up
No
no longer a termination reason
applicationWillEnterForeground:
App Relaunched (e.g. Springboard, App Switcher or URL)
Background Run Loop
SDK >= 4
No
Yes
Yes
App Suspended in RAM
No
info.plist setting or task completion
No
UIApplication-ExitsOnSuspend
"opt out"
Yes
applicationWillTerminate:
OS needs to reclaim memory
Background
kill
App Ended
App has left the RAM



created by Oliver Drobnik, oliver@drobnik.com, Twitter dr_touch

UIApplication Background Task

参考资料：

- UIApplication 深入学习

UIView 表示屏幕上的一块矩形区域，负责渲染区域的内容，并且响应该区域内发生的触摸事件。它在iOS App中占有绝对重要的地位，因为iOS中几乎所有可视化控件都是 UIView 的子类。

UIView 可以负责以下几种任务：

- 绘制和动画
- 布局和子视图管理
- 事件处理

绘制和动画

视图绘制

UIView 是按需绘制的，当整个视图或者视图的一部分由于布局变化，变成可见的，系统会要求视图进行绘制。对于那些需要使用 UIKit 或者 CoreGraphics 进行自定义绘制的视图，系统会调用 `drawRect:` 方法进行绘制。

当视图内容发生变化时，需要调用 `setNeedsDisplay` 或者 `setNeedsDisplayInRect:` 方法，告诉系统该重新绘制这个视图了。调用这个方法之后，系统会在下一个绘制周期更新这个视图的内容。由于系统要等到下一个绘制周期才真正进行绘制，可以一次性对多个视图调用 `setNeedsDisplay` ，它们会同时被更新。

视图的几何属性

视图有 frame，center，bounds 等几个基本几何属性，其中:

- frame 使用的最多，其坐标位置都是相对于父视图的，可以用于确定本视图在父视图中的位置和其自身的大小
- center 的坐标位置也是相对于父视图的，通常用于移动，旋转等动画操作
- bounds 是相对于自身的，通常情况下就是（0,0,width,height)， bounds 的含义可以认为是当前 view 被允许绘制的范围

视图的 **ContentMode**

视图在初次绘制完成后，系统会对绘制结果进行快照，之后尽可能地使用快照，避免重新绘制。如果视图的几何属性发生改变，系统会根据视图的 contentMode 来决定如何改变显示效果。

默认的 contentMode 是 `UIViewContentModeScaleToFill` ，系统会拉伸当前的快照，使其符合新的 frame 尺寸。大部分 contentMode 都会对当前的快照进行拉伸或者移动等操作。如果需要重新绘制，可以把 contentMode 设置为 `UIViewContentModeRedraw` ，强制视图在改变大小之

类的操作时调用 `drawRect:` 重绘。

动画

可以以动画的形式改变视图的下面这些属性，只需要告诉系统动画开始和结束时的数值，系统会自动处理中间的过渡过程。

- frame
- bounds
- center
- transform
- alpha
- backgroundColor
- contentStretch

布局和子视图管理

除了提供视图本身的内容之外，一个视图也可以表现得像一个容器。当一个视图包含其他视图时，两个视图之间就创建了一个父子关系。在这个关系中子视图被称为 subView ，父视图被称为 superView 。一个视图可以包含多个子视图，它们被存放在这个视图的 `subviews` 数组里。添加，删除，以及操作这些子视图的相对位置的函数如下：

- `addSubview:`
- `insertSubview:...`
- `bringSubviewToFront:`
- `sendSubviewToBack:`
- `exchangeSubviewAtIndex:withSubviewAtIndex:`
- `removeFromSuperview` （子视图调用）

AutoResizing 和 Constraint

当一个视图的大小改变时，它的子视图的位置和大小也需要相应地改变。UIView 支持自动布局，也可以手动对子视图进行布局。

当下列这些事件发生时，需要进行布局操作：

- 视图的 bounds 大小改变#
- 用户界面旋转，通常会导致根视图控制器的大小改变
- 视图的 layer 层的 Core Animation sublayers 发生改变
- 程序调用视图的 `setNeedsLayout` 或 `layoutIfNeeded` 方法
- 程序调用视图 layer 的 `setNeedsLayout` 方法

Auto Resizing

视图的 `autoresizesSubviews` 属性决定了在视图大小发生变化时，如何自动调节子视图。

可以使用的掩码如下：

- UIViewAutoresizingNone
- UIViewAutoresizingFlexibleHeight
- UIViewAutoresizingFlexibleWidth
- UIViewAutoresizingFlexibleLeftMargin
- UIViewAutoresizingFlexibleRightMargin
- UIViewAutoresizingFlexibleBottomMargin
- UIViewAutoresizingFlexibleTopMargin

可以通过位运算符将它们组合起来，例如 `UIViewAutoresizingFlexibleHeight|UIViewAutoresizingFlexibleWidth` 。

Constraint

Constraint 是另一种用于自动布局的方法。本质上，Constraint 就是对 UIView 之间两个属性的一个约束：

```
attribute1 == multiplier × attribute2 + constant
```

其中方程两边不一定是等于关系，也可以是大于等于之类的关系。

Constraint 比 AutoResizing 更加灵活和强大，可以实现复杂的子视图布局。

自定义 layout

UIView 当中提供了一个 `layoutSubviews` 函数，UIView 的子类可以重载这个函数，以实现更加复杂和精细的子 View 布局。

苹果文档专门强调了，应该只在上面提到的 Autoresizing 和 Constraint 机制不能实现所需要的效果时，才使用 `layoutSubviews` 。而且，layoutSubviews 方法只能被系统触发调用，程序员不能手动直接调用该方法。

那么 layoutSubviews 方法具体调用的时机有哪些呢？在 stackoverflow 的这个答案里有所讨论，具体有下面几种情况：

1. 在父 view 的 autoresize mask 为 ON 的情况下，addSubview 会导致被 add 的 view 调用 layoutSubviews, 同时 add 的 target view 以及它所有的子 view 都会被调用。
2. setFrame 当新的 frame 和 旧的不同时（即 view 的大小改变时）会调用 layoutSubviews
3. 滚动一个 UIScollView 会导致这个 scrollView 以及它的父 View 调用 layoutSubviews
4. 旋转设备会导致当前所响应的 ViewController 的主 View 调用 layoutSubviews
5. 改变 View 的 size 会导致父 View 调用 layoutSubviews

6. removeFromSuperview 也会导致父 View 调用 layoutSubviews

重载 layoutSubviews 可以让我们实现更复杂的布局效果，这篇博客里以 RDVTabBarController为例进行了简单介绍。

事件处理

UIView 是 UIResponder 的子类，可以响应触控事件。

通常可以使用 `addGestureRecognizer:` 添加手势识别器来响应触控事件，如果需要手动处理，则按需要重载 UIView 中的下面四个函数：

- `touchesBegan:withEvent:`
- `touchesMoved:withEvent:`
- `touchesEnded:withEvent:`
- `touchesCancelled:withEvent:`

参考资料

- UIView详解
- UIView你知道多少
- http://stackoverflow.com/questions/728372/when-is-layoutsubviews-called

UIViewController（视图控制器），顾名思义，是 MVC 设计模式中的控制器部分。UIViewController 在 UIKit 中主要功能是用于控制画面的切换，其中的 `view` 属性（UIView 类型）管理整个画面的外观。

UIViewController 生命周期

ViewController 生命周期的第一步是初始化。不过具体调用的方法还有所不同。如果使用 StoryBoard 来创建 ViewController，我们不需要显式地去初始化，Storyboard 会自动使用 `initWithCoder:` 进行初始化。如果不使用 StoryBoard，我们可以使用 `init:` 函数进行初始化，`init:` 函数在实现过程中还会调用 `initWithNibName:bundle:`。我们应该尽量避免在 VC 外部调用 `initWithNibName:bundle:`，而是把它放在 VC 的内部（参考这里）。

初始化完成后，VC 的生命周期会经过下面几个函数：

- (void)loadView
- (void)viewDidLoad
- (void)viewWillAppear
- (void)viewWillLayoutSubviews
- (void)viewDidLayoutSubviews
- (void)viewDidAppear
- (void)viewWillDisappear
- (void)viewDidDisappear

假设现在有一个 AViewController(简称 Avc) 和 BViewController (简称 Bvc)，通过 navigationController 的 push 实现 Avc 到 Bvc 的跳转，下面是各个方法的执行执行顺序：

```
1. A viewDidLoad
2. A viewWillAppear
3. A viewDidAppear
4. B viewDidLoad
5. A viewWillDisappear
6. B viewWillAppear
7. A viewDidDisappear
8. B viewDidAppear
```

如果再从 Bvc 跳回 Avc，会产生下面的执行顺序：

```
1. B viewWillDisappear
2. A viewWillAppear
3. B viewDidDisappear
4. A viewDidAppear
```

可见 viewDidLoad 只会调用一次，再第二次跳回 Avc 的时候，AViewController 仍然存在于内存中，也就不需要 load 了。

注意上面的生命周期中都没有提到有关 ViewController 销毁的内容，在 iOS 4 & 5 中 ViewController 中有一个 `viewDidUnload` 方法。当内存不足，应用收到 Memory warning 时，系统会自动调用当前没在界面上的 ViewController 的 viewDidUnload 方法。 通常情况下，这些未显示在界面上的 ViewController 是 UINavigationController Push 栈中未在栈顶的 ViewController，以及 UITabBarViewController 中未显示的子 ViewController。这些 View Controller 都会在 Memory Warning 事件发生时，被系统自动调用 viewDidUnload 方法。

从 iOS 6 开始，viewDidUnload 方法被废弃掉了，应用受到 memory warning 时也不会再调用 viewDidUnload 方法。我们可以通过重载 `- (void)didReceiveMemoryWarning` 和 `-(void)dealloc` 来进行清理工作。

参考资料

1. UIViewController生命周期方法执行顺序
2. http://blog.devtang.com/blog/2013/05/18/goodbye-viewdidunload/

Core Animation

注：示例中部分代码的完整版可以在这里找到。

UIView Animation

简单动画

对于 UIView 上简单的动画，iOS 提供了很方便的函数：

```
+ animateWithDuration:animations:
```

第一个参数是动画的持续时间，第二个参数是一个 block，在 `animations` block 中对 UIView 的属性进行调整，设置 UIView 动画结束后最终的效果，iOS 就会自动补充中间帧，形成动画。

可以更改的属性有:

- frame
- bounds
- center
- transform
- alpha
- backgroundColor
- contentStretch

这些属性大都是 View 的基本属性，下面是一个例子，这个例子中的动画会同时改变 View 的 `frame` ， `backgroundColor` 和 `alpha` ：

```
[UIView animateWithDuration:2.0 animations:^{
    myView.frame = CGRectMake(50, 200, 200, 200);
    myView.backgroundColor = [UIColor blueColor];
    myView.alpha = 0.7;
}];
```

其中有一个比较特殊的 `transform` 属性，它的类型是 `CGAffineTransform` ，即 2D 仿射变换，这是个数学中的概念，用一个三维矩阵来表述 2D 图形的矢量变换。用 `transform` 属性对 View 进行：

- 旋转
- 缩放
- 其他自定义 2D 变换

iOS 提供了下面的函数可以创建简单的 2D 变换：

- `CGAffineTransformMakeScale`
- `CGAffineTransformMakeRotation`
- `CGAffineTransformMakeTranslation`

例如下面的代码会将 View 缩小至原来的 1/4 大小：

```
[UIView animateWithDuration:2.0 animations:^{
    myView.transform = CGAffineTransformMakeScale(0.5, 0.5);
}];
```

调节参数

完整版的 animate 函数其实是这样的：

```
+ animateWithDuration:delay:options:animations:completion:
```

可以通过 `delay` 参数调节让动画延迟产生，同时还一个 `options` 选项可以调节动画进行的方式。可用的 `options` 可分为两类：

控制过程

例如 `UIViewAnimationOptionRepeat` 可以让动画反复进行，`UIViewAnimationOptionAllowUserInteraction` 可以让允许用户对动画进行过程中同 View 进行交互（默认是不允许的）

控制速度

动画的进行速度可以用速度曲线来表示（参考这里），提供的选项例如 `UIViewAnimationOptionCurveEaseIn` 是先慢后快，`UIViewAnimationOptionCurveEaseOut` 是先快后慢。

不同的选项直接可以通过“与”操作进行合并，同时使用，例如:

```
UIViewAnimationOptionRepeat | UIViewAnimationOptionAllowUserInteraction
```

关键帧动画

上面介绍的动画中，我们只能控制开始和结束时的效果，然后由系统补全中间的过程，有些时候我们需要自己设定若干关键帧，实现更复杂的动画效果，这时候就需要关键帧动画的支持了。下面是一个示例：

```
[UIView animateKeyframesWithDuration:2.0 delay:0.0 options:UIViewKeyframeAnimationOpti
onRepeat | UIViewKeyframeAnimationOptionAutoreverse animations:^{
    [UIView addKeyframeWithRelativeStartTime:0.0 relativeDuration:0.5 animations:^{
        self.myView.frame = CGRectMake(10, 50, 100, 100);
    }];
    [UIView addKeyframeWithRelativeStartTime: 0.5 relativeDuration:0.3 animations:^{
        self.myView.frame = CGRectMake(20, 100, 100, 100);
    }];
    [UIView addKeyframeWithRelativeStartTime:0.8 relativeDuration:0.2 animations:^{
        self.myView.transform = CGAffineTransformMakeScale(0.5, 0.5);
    }];
} completion:nil];
```

这个例子添加了三个关键帧，在外面的 `animateKeyframesWithDuration` 中我们设置了持续时间为 2.0 秒，这是真实意义上的时间，里面的 `startTime` 和 `relativeDuration` 都是相对时间。以第一个为例，`startTime` 为 0.0，`relativeTime` 为 0.5，这个动画会直接开始，持续时间为 2.0 X 0.5 = 1.0 秒，下面第二个的开始时间是 0.5，正好承接上一个结束，第三个同理，这样三个动画就变成连续的动画了。

View 的转换

iOS 还提供了两个函数，用于进行两个 View 之间通过动画换场：

```
+ transitionWithView:duration:options:animations:completion:
+ transitionFromView:toView:duration:options:completion:
```

需要注意的是，换场动画会在这两个 View 共同的父 View 上进行，在写动画之前，先要设计好 View 的继承结构。

同样，View 之间的转换也有很多选项可选，例如 `UIViewAnimationOptionTransitionFlipFromLeft` 从左边翻转，`UIViewAnimationOptionTransitionCrossDissolve` 渐变等等。

CALayer Animation

UIView 的动画简单易用，但是能实现的效果相对有限，上面介绍的 UIView 的几种动画方式，实际上是对底层 CALayer 动画的一种封装。直接使用 CALayer 层的动画方法可以实现更多高级的动画效果。

注意：使用 CALayer 动画之前，首先需要引入 QuartzCore.framework。

基本动画（CABasicAnimation）

CABasicAnimation 用于创建一个 CALayer 上的基本动画效果，下面是一个例子：

```
CABasicAnimation *animation = [CABasicAnimation animationWithKeyPath:@"position.x"];
animation.toValue = @200;
animation.duration = 0.8;
animation.repeatCount = 5;
animation.beginTime = CACurrentMediaTime() + 0.5;
animation.fillMode = kCAFillModeRemoved;
[self.myView.layer addAnimation:animation forKey:nil];
```

KeyPath

这里我们使用了 `animationWithKeyPath` 这个方法来改变 layer 的属性，可以使用的属性有很多，具体可以参考这里和这里。其中很多属性在前面介绍的 UIView 动画部分我们也看到过，进一步验证了 UIView 的动画方法是对底层 CALayer 的一种封装。

需要注意的一点是，上面我们使用了 `position` 属性，layer 的这个 `position` 属性和 View 的 `frame` 以及 `bounds` 属性都不相同，而是和 Layer 的 `anchorPoint` 有关，可以由下面的公式计算得到：

```
position.x = frame.origin.x + 0.5 * bounds.size.width；
position.y = frame.origin.y + 0.5 * bounds.size.height；
```

关于 `anchorPoint` 和 `position` 属性的以及具体计算的原理可以参考这篇文章。

属性

CABasicAnimation 的属性有下面几个：

- beginTime
- duration
- fromValue
- toValue
- byValue
- repeatCount
- autoreverses
- timingFunction

可以看到，其中 beginTime，duration，repeatCount 等属性和上面在 UIView 中使用到的 duration，UIViewAnimationOptionRepeat 等选项是相对应的，不过这里的选项能够提供更多的扩展性。

需要注意的是 `fromValue` ，`toValue` ，`byValue` 这几个选项，支持的设置模式有下面几种：

- 设置 fromValue 和 toValue：从 fromValue 变化到 toValue
- 设置 fromValue 和 byValue：从 fromValue 变化到 fromValue + byValue
- 设置 byValue 和 toValue：从 toValue - byValue 变化到 toValue
- 设置 fromValue： 从 fromValue 变化到属性当前值

- 设置 toValue：从属性当前值变化到 toValue
- 设置 byValue：从属性当前值变化到属性当前值 + toValue

看起来挺复杂，其实概括起来基本就是，如果某个值不设置，就是用这个属性当前的值。

另外，可以看到上面我们使用的:

```
animation.toValue = @200;
```

而不是直接使用 200，因为 `toValue` 之类的属性为 `id` 类型，或者像这样使用 @ 符号，或者使用：

```
animation.toValue = [NSNumber numberWithInt:200];
```

最后一个比较有意思的是 `timingFunction` 属性，使用这个属性可以自定义动画的运动曲线（节奏，pacing），系统提供了五种值可以选择：

- kCAMediaTimingFunctionLinear 线性动画
- kCAMediaTimingFunctionEaseIn 先快后慢
- kCAMediaTimingFunctionEaseOut 先慢后快
- kCAMediaTimingFunctionEaseInEaseOut 先慢后快再慢
- kCAMediaTimingFunctionDefault 默认，也属于中间比较快

此外，我们还可以使用 [CAMediaTimingFunction functionWithControlPoints] 方法来自定义运动曲线，这个网站提供了一个将参数调节可视化的效果，关于动画时间系统的具体介绍可以参考这篇文章。

关键帧动画（CAKeyframeAnimation）

同 UIView 中的类似，CALayer 层也提供了关键帧动画的支持，CAKeyFrameAnimation 和 CABasicAnimation 都继承自 CAPropertyAnimation，因此它有具有上面提到的那些属性，此外，CAKeyFrameAnimation 还有特有的几个属性。

values 和 keyTimes

使用 `values` 和 `keyTimes` 可以共同确定一个动画的若干关键帧，示例代码如下：

```
CAKeyframeAnimation *anima = [CAKeyframeAnimation animationWithKeyPath:@"transform.rotation"];//在这里@"transform.rotation"==@"transform.rotation.z"
NSValue *value1 = [NSNumber numberWithFloat:-M_PI/180*4];
NSValue *value2 = [NSNumber numberWithFloat:M_PI/180*4];
NSValue *value3 = [NSNumber numberWithFloat:-M_PI/180*4];
anima.values = @[value1,value2,value3];
// anima.keyTimes = @[@0.0, @0.5, @1.0];
anima.repeatCount = MAXFLOAT;

[_demoView.layer addAnimation:anima forKey:@"shakeAnimation"];
```

可以看到上面这个动画共有三个关键帧，如果没有指定 `keyTimes` 则各个关键帧会平分整个动画的时间(duration)。

path

使用 path 属性可以设置一个动画的运动路径，注意 path 只对 CALayer 的 anchorPoint 和 position 属性起作用，另外如果你设置了 path ，那么 values 将被忽略。

```
CAKeyframeAnimation *anima = [CAKeyframeAnimation animationWithKeyPath:@"position"];
UIBezierPath *path = [UIBezierPath bezierPathWithOvalInRect:CGRectMake(SCREEN_WIDTH/2-
100, SCREEN_HEIGHT/2-100, 200, 200)];
anima.path = path.CGPath;
anima.duration = 2.0f;
[_demoView.layer addAnimation:anima forKey:@"pathAnimation"];
```

组动画（CAAnimationGroup)

组动画可以将一组动画组合在一起，所有动画对象可以同时运行，示例代码如下：

```
CAAnimationGroup *group = [[CAAnimationGroup alloc] init];
CABasicAnimation *animationOne = [CABasicAnimation animationWithKeyPath:@"transform.sc
ale"];

animationOne.toValue = @2.0;
animationOne.duration = 1.0;

CABasicAnimation *animationTwo = [CABasicAnimation animationWithKeyPath:@"position.x"]
;
animationTwo.toValue = @400;
animationTwo.duration = 1.0;

[group setAnimations:@[animationOne, animationTwo]];
[self.myView.layer addAnimation:group forKey:nil];
```

需要注意的是，一个 group 组内的某个动画的持续时间（duration），如果超过了整个组的动画持续时间，那么多出的动画时间将不会被展示。例如一个 group 的持续时间是 5s，而组内一个动画持续时间为 10s ，那么这个 10s 的动画只会展示前 5s 。

切换动画（CATransition）

CATransition 可以用于 View 或 ViewController 直接的换场动画：

```
self.myView.backgroundColor = [UIColor blueColor];
CATransition *trans = [CATransition animation];
trans.duration = 1.0;
trans.type = @"push";

[self.myView.layer addAnimation:trans forKey:nil];

// 这句放在下面也可以
// self.myView.backgroundColor = [UIColor blueColor];
```

为什么改变颜色放在前后都可以呢？具体的解释可以参考 SO 上的这个回答。简单来说就是动画和绘制之间并不冲突。

更高级的动画效果

CADisplayLink

CADisplayLink 是一个计时器对象，可以周期性的调用某个 selecor 方法。相比 NSTimer ，它可以让我们以和屏幕刷新率同步的频率（每秒60次）来调用绘制函数，实现界面连续的不停重绘，从而实现动画效果。

示例代码（修改自这里)：

```
#import "BlockView.h"

@implementation BlockView

- (void)startAnimationFrom:(CGFloat)from To:(CGFloat)to
{
    self.from = from;
    self.to = to;
    if (self.displayLink == nil) {
        self.displayLink = [CADisplayLink displayLinkWithTarget:self selector:@selector
(tick:)];
        [self.displayLink addToRunLoop:[NSRunLoop currentRunLoop]
                               forMode:NSDefaultRunLoopMode];
    }
}

// 重复调用这个方法以重绘整个 View
- (void)tick:(CADisplayLink *)displayLink
{
    [self setNeedsDisplay];
}

- (void)endAnimation
{
    [self.displayLink invalidate];
    self.displayLink = nil;
}

- (void)drawRect:(CGRect)rect
{
    CALayer *layer = self.layer.presentationLayer;
    CGFloat progress = 1 - (layer.position.y - self.to) / (self.from - self.to);
    CGFloat height = CGRectGetHeight(rect);
    CGFloat deltaHeight = height / 2 * (0.5 - fabs(progress - 0.5));
    CGPoint topLeft = CGPointMake(0, deltaHeight);
    CGPoint topRight = CGPointMake(CGRectGetWidth(rect), deltaHeight);
    CGPoint bottomLeft = CGPointMake(0, height);
    CGPoint bottomRight = CGPointMake(CGRectGetWidth(rect), height);
    UIBezierPath* path = [UIBezierPath bezierPath];
    [[UIColor blueColor] setFill];
    [path moveToPoint:topLeft];
    [path addQuadCurveToPoint:topRight controlPoint:CGPointMake(CGRectGetMidX(rect), 0
)];
    [path addLineToPoint:bottomRight];
    [path addQuadCurveToPoint:bottomLeft controlPoint:CGPointMake(CGRectGetMidX(rect),
 height - deltaHeight)];
    [path closePath];
    [path fill];
}

@end
```

UIDynamicAnimator

UIDynamicAnimator 是 iOS 7 引入的一个新类，可以创建出具有物理仿真效果的动画，具体提供了下面几种物理仿真行为：

- UIGravityBehavior：重力行为
- UICollisionBehavior：碰撞行为
- UISnapBehavior：捕捉行为
- UIPushBehavior：推动行为
- UIAttachmentBehavior：附着行为
- UIDynamicItemBehavior：动力元素行为

示例代码如下（来自这里)

```
self.animator = [[UIDynamicAnimator alloc] initWithReferenceView:self.view];

UIGravityBehavior* gravityBehavior = [[UIGravityBehavior alloc] initWithItems:@[self.m
yView]];
[self.animator addBehavior:gravityBehavior];

UICollisionBehavior* collisionBehavior = [[UICollisionBehavior alloc] initWithItems:@[
self.myView]];
collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;
[self.animator addBehavior:collisionBehavior];
```

可以发现这段代码和我们之前写的动画代码有很大不同，在这里 behavior 是用于控制 View 行为的，我们做的操作是把各种不同的 behavior 加到 animator 中。这段代码实现了 View 因为“重力”原因“掉到”地上，落地的同时还有一个碰撞效果。

CAEmitterLayer

CAEmitterLayer 是 Core Animation 提供的一个粒子发生器系统，可以用于创建各种粒子动画，例如烟雾，焰火等效果。

CAEmitterLayer 需要调节的参数很多，可以实现的效果也非常炫酷，具体可参考下面几个网址：

- http://enharmonichq.com/tutorial-particle-systems-in-core-animation-with-caemitterlayer/#prettyPhoto/0/
- https://www.invasivecode.com/weblog/caemitterlayer-and-the-ios-particle-system-lets/?doing_wp_cron=1438657800.4759559631347656250000

LTMorphingLabel 这个项目使用 CAEmitterLayer 实现了各种高端炫酷掉渣天的效果，大家想学习的话可以去看看它的代码。

Cocoa 网络编程

Cocoa 中网络编程层次结构分为三层，自上而下分别是：

- Cocoa 层：NSURL，Bonjour，Game Kit，WebKit
- Core Foundation 层：基于 C 的 CFNetwork 和 CFNetServices
- OS 层:基于 C 的 BSD socket

这里主要介绍处于 Cocoa 层的基于 NSURL 的一系列方法。在 iOS7 之前，主要使用的网络编程 API 是 NSURLConnection 一族的类，在 iOS7 之后苹果引入了 NSURLSession 类族，用于替代 NSURLConnection。

注意：在 **Xcode 7 / iOS 9.0** 中苹果正式废弃了 **NSURLConnection** 系列 **API**，并建议开发者尽快迁移到 **NSURLSession**。因此下面有关 **NSURLConnection** 的内容仅作为参考使用。

NSURLConnection

CoreFoundation 中提供了一个类 NSURLConnection ，用于处理用户的网络请求，NSURLConnection 基本可以满足我们大多数的网络请求操作。NSURLConnection 本身并不能单独使用，需要与一族网络通信有关的类进行协同工作，包括 NSURLRequest, NSURLResponse，NSURLCache 等等。

基本的请求操作

同步请求，使用 **sendAsynchronousRequest** 方法

```
+ (NSData *)sendSynchronousRequest:(NSURLRequest *)request
                 returningResponse:(NSURLResponse **)response
                             error:(NSError **)error;
```

这个同步请求是阻塞的，并且不可以中途 cancel 掉。我们可以将同步请求放到主线程之外的线程中，执行效果也会类似于异步，比如放到 GCD 的 dispatch_async 里面执行。

异步请求，使用 **sendAsynchronousRequest**

```
+ (void)sendAsynchronousRequest:(NSURLRequest*) request
                          queue:(NSOperationQueue*) queue
              completionHandler:(void (^)(NSURLResponse* response, NSData* data, NSErr
or* connectionError)) handler;
```

这个异步请求是非阻塞的，异步执行后把结果通过 block 回调回来，不能中途 cancel 掉

异步请求，使用委托

首先初始化请求：

```
- (id)initWithRequest:(NSURLRequest *)request delegate:(id)delegate;
```

然后根据需要在 delegate 类(NSURLConnectionDataDelegate协议)里面实现下列代理函数，获取异步请求的返回的数据与结果

```
- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)r
esponse
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
```

这个异步请求是非阻塞的，异步执行后把返回的数据与结果通过 delegate 函数回调回来，可以使用 cancel 中途取消。

将请求放到后台线程

上面提到的 NSURLConnection 的异步方法实际上还是跑在主线程当中，在主线程中执行网络操作会带来两个问题：

1. 尽管在网络连接过程中不会对主线程造成阻塞，但是 delegate 的回调方法还是在主线程中执行的。如果我们在回调方法中（特别是 completion 回调）中进行了大量的耗时操作，仍然会造成主线程的阻塞。
2. NSURLConnection 默认会跑在当前的 runloop 中，并且跑在 Default Mode，当用户执行滚动的 UI 操作时会发生 runloop mode 的切换，也就导致了 NSURLConnection 不能及时执行和完成回调。

为了解决这些问题，我们可以让整个 NSURLConnection 都在后台线程中执行。

怎么做？

简单地把 `start` 函数放到后台的 queue 中是不行的，像下面这样：

```
dispatch_async(connectionQueue, ^{
        NSMutableURLRequest *request = [[NSMutableURLRequest alloc] init];
        [request setURL:[NSURL URLWithString:[NSString stringWithFormat:someURL]]];

        NSURLConnection *connection = [[NSURLConnection alloc] initWithRequest:request
 delegate:self]; // 没有设置 startImmediately 为 NO，会立即开始
        //[connection start]; 这一句没有必要写，写了也一样不能 work。
});
```

因为 dispatch_async 开出的线程中，默认 runloop 没有执行，因此线程会立即结束，来不及调用回调方法。我们可以添加代码让 runloop 跑起来：

```
dispatch_async(connectionQueue, ^{
        NSMutableURLRequest *request = [[NSMutableURLRequest alloc] init];
        [request setURL:[NSURL URLWithString:[NSString stringWithFormat:someURL]]];

        NSURLConnection *connection = [[NSURLConnection alloc] initWithRequest:request
 delegate:self];
        [[NSRunLoop currentRunLoop] run];
});
```

这样回调函数才能够被调用，但是这样又带来一个问题，这个线程中 runloop 会一直跑着，导致这个线程也一直不结束，为了让所在线程在完成任务时正确释放掉，我们可以这样做：

```
dispatch_async(connectionQueue, ^{
        NSMutableURLRequest *request = [[NSMutableURLRequest alloc] init];
        [request setURL:[NSURL URLWithString:[NSString stringWithFormat:someURL]]];

        NSURLConnection *connection = [[NSURLConnection alloc] initWithRequest:request
 delegate:self];
        while(!self.finished) {
            [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode beforeDate:[NSDate
 distantFuture]];
        }
    });
```

然后在 finish 回调中执行：

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    self.finish = YES;
}
```

这样的实现实际上是有些 dirty 的，引入了一个死循环来判断是否应该终止 loop。看起来 GCD 并不适合和 NSURLConnection 一起工作。

除了 GCD 之外就没有别的办法了吗？幸好，苹果还提供了下面两种方法：

scheduleInRunLoop:forMode:

这个函数可以让我们指定 NSURLConnection 跑在某个 runloop：

```
NSRunLoop* runLoop = [NSRunLoop currentRunLoop];
[runLoop addPort:[NSMachPort port] forMode:NSDefaultRunLoopMode]; // 添加 inputSource，
让 runloop 保持 alive
[self.connection scheduleInRunLoop:runLoop
                           forMode:NSDefaultRunLoopMode];
[self.connection start];
[runLoop run];
```

这样，我们把它加到任意的有 Runloop 的线程中（其实 Cocoa 的线程都是自带 runloop 的，不过没有打开）都可以正常工作了，加到 NSOperationQueue 中也是可以的。

知名的开源网络库 AFNetworking 就是这么做的，代码参考这里。

注意一点，这样做的话， NSURLConnection 任务所在的线程是永远不会退出的，为了让它正确退出，可以在请求完成时结束掉 runloop：

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    CFRunLoopStop(CFRunLoopGetCurrent());
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
    CFRunLoopStop(CFRunLoopGetCurrent());
}
```

AFNetworking 中负责响应回调的线程，就是通过 Runloop 来保持永不退出的，一直在后台负责响应回调。

setDelegateQueue:

更简单的方法是直接使用这个函数，直接使用 NSOperationQueue 来管理我们的 Connection：

```
NSURLConnection *connection = [[NSURLConnection alloc] initWithRequest:aURLRequest
                                                              delegate:self
                                                      startImmediately:NO];
[connection setDelegateQueue:[[NSOperationQueue alloc] init]];
[connection start];
```

如果我们不需要太多自定义功能，这个函数也完全够用了，不需要配置 runloop，不需要担心线程不会正常退出的问题，可以让我们专注于业务代码的编写。

注意上面提到的这两个函数只能取其中一个，如果同时用了两个会报错。

NSURLSession

http://objccn.io/issue-5-4/

参考资料

- [深入浅出Cocoa]iOS网络编程系列
- Cocoa网络编程总结之NSURLConnection
- http://iosdevelopmentjournal.com/blog/2013/01/27/running-network-requests-in-the-background/
- https://stackoverflow.com/questions/8941353/ios-dispatch-async-and-nsurlconnection-delegate-functions-not-being-called/13733626#13733626
- https://satanwoo.github.io/2015/09/11/A-New-Start/
- https://stackoverflow.com/questions/1728631/asynchronous-request-to-the-server-from-

background-thread

- https://stackoverflow.com/questions/1363787/is-it-safe-to-call-cfrunloopstop-from-another-thread
- http://www.dribin.org/dave/blog/archives/2009/05/05/concurrent_operations/
- http://nshipster.com/nsoperation/```objectivec

Cocoa 并发编程

iOS 中的多线程，是 Cocoa 框架下的多线程，通过 Cocoa 的封装，可以让我们更为方便的进行多线程编程。

在介绍 Cocoa 并发编程之前，我们先理清会提到的几个术语：

- 线程：就是我们通常提到的线程，在进程中可以用线程去执行一些主进程之外的代码。OS X 中线程的实现基于 POSIX 的 pthread API。
- 进程：也是我们通常意义上提到的进程，一个正在执行中的程序实体，可以产生多个线程
- 任务：一个抽象的概念，用于表示一系列需要完成的工作

Cocoa 中封装了 NSThread, NSOperation, GCD 三种多线程编程方式，他们各有所长。

- NSThread

  NSThread 是一个控制线程执行的对象，通过它我们可以方便的得到一个线程并控制它。NSThread 的线程之间的并发控制，是需要我们自己来控制的，可以通过 NSCondition 实现。它的缺点是需要自己维护线程的生命周期和线程的同步和互斥等，优点是轻量，灵活。

- NSOperation

  NSOperation 是一个抽象类，它封装了线程的细节实现，不需要自己管理线程的生命周期和线程的同步和互斥等。只是需要关注自己的业务逻辑处理，需要和 NSOperationQueue 一起使用。使用 NSOperation 时，你可以很方便的设置线程之间的依赖关系。这在略微复杂的业务需求中尤为重要。

- GCD

  GCD(Grand Central Dispatch) 是 Apple 开发的一个多核编程的解决方法。在 iOS4.0 开始之后才能使用。GCD 是一个可以替代 NSThread 的很高效和强大的技术。当实现简单的需求时，GCD 是一个不错的选择。

在现代 Objective-C 中，苹果已经不推荐使用 NSThread 来进行并发编程，而是推荐使用 GCD 和 NSOperation，具体的迁移文档参见 Migrating Away from Threads。下面我们对 GCD 和 NSOperation 的用法进行简单介绍。

Grand Central Dispatch(GCD)

Grand Central Dispatch(GCD) 是苹果在 Mac OS X 10.6 以及 iOS 4.0 开始引入的一个高性能并发编程机制，底层实现的库名叫 libdispatch。由于它确实很好用，libdispatch 已经被移植到了 FreeBSD 上，Linux 上也有 port 过去的 libdispatch 实现。

GCD 这么受大家欢迎，它具体好用在哪里呢？GCD 主要的功劳在于把底层的实现隐藏起来，提供了很简洁的面向“任务” 的编程接口，让程序员可以专注于代码的编写。GCD 底层实现仍然依赖于线程，但是使用 GCD 时完全不需要考虑下层线程的有关细节（创建任务比创建线程简单得多），GCD 会自动对任务进行调度，以尽可能地利用处理器资源。

想要了解 GCD，首先要了解下面几个概念：

- Dispatch Queue：Dispatch Queue 顾名思义，是一个用于维护任务的队列，它可以接受任务（即可以将一个任务加入某个队列）然后在适当的时候执行队列中的任务。
- Dispatch Sources：Dispatch Source 允许我们把任务注册到系统事件上，例如 socket 和文件描述符，类似于 Linux 中 epoll 的作用
- Dispatch Groups：Dispatch Groups 可以让我们把一系列任务加到一个组里，组中的每一个任务都要等待整个组的所有任务都结束之后才结束，类似 pthread_join 的功能
- Dispatch Semaphores：这个更加顾名思义，就是大家都知道的信号量了，可以让我们实现更加复杂的并发控制，防止资源竞争

这些东西中最经常用到的是 Dispatch Queue。之前提到 Dispatch Queue 就是一个类似队列的数据结构，而且是 FIFO(First In, First Out)队列，因此任务开始执行的顺序，就是你把它们放到 queue 中的顺序。GCD 中的队列有下面三种：

1. Serial （串行队列） 串行队列中任务会按照添加到 queue 中的顺序一个一个执行。串行队列在前一个任务执行之前，后一个任务是被阻塞的，可以利用这个特性来进行同步操作。

   我们可以创建多个串行队列，这些队列中的任务是串行执行的，但是这些队列本身可以并发执行。例如有四个串行队列，有可能同时有四个任务在并行执行，分别来自这四个队列。

2. Concurrent（并行队列） 并行队列，也叫 global dispatch queue，可以并发地执行多个任务，但是任务开始的顺序仍然是按照被添加到队列中的顺序。具体任务执行的线程和任务执行的并发数，都是由 GCD 进行管理的。

   在 iOS 5 之后，我们可以创建自己的并发队列。系统已经提供了四个全局可用的并发队列，后面会讲到。

3. Main Dispatch Queue（主队列） 主队列是一个全局可见的串行队列，其中的任务会在主线程中执行。主队列通过与应用程序的 runloop 交互，把任务安插到 runloop 当中执行。因为主队列比较特殊，其中的任务确定会在主线程中执行，通常主队列会被用作同步的作用。

获取队列

按照上面提到的三种队列，我们有对应的三种获取队列的方式：

1. 串行队列 系统默认并不提供串行队列，需要我们手动创建：

```
dispatch_queue_t queue;
queue = dispatch_queue_create("com.example.MyQueue", NULL); // OS X 10.7 和 iOS 4
.3 之前
queue = dispatch_queue_create("com.example.MyQueue",  DISPATCH_QUEUE_SERIAL); //
之后
```

2. 并行队列 系统默认提供了四个全局可用的并行队列，其优先级不同，分别为 `DISPATCH_QUEUE_PRIORITY_HIGH` ，`DISPATCH_QUEUE_PRIORITY_DEFAULT` ，`DISPATCH_QUEUE_PRIORITY_LOW` ，`DISPATCH_QUEUE_PRIORITY_BACKGROUND` ，优先级依次降低。优先级越高的队列中的任务会更早执行：

```
dispatch_queue_t aQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAU
LT, 0);
```

   当然我们也可以创建自己的并行队列：

```
queue = dispatch_queue_create("com.example.MyQueue", DISPATCH_QUEUE_CONCURRENT);
```

   不过一般情况下我们使用系统提供的 Default 优先级的 queue 就足够了。

   更新：在 iOS8+ 和 OS X 10.10+ 中苹果引入了新的 QOS 类别，具体的几个类别如下：

   - `QOS_CLASS_USER_INTERACTIVE`
   - `QOS_CLASS_USER_INITIATED`
   - `QOS_CLASS_UTILITY`
   - `QOS_CLASS_BACKGROUND`

     在支持的平台上，推荐使用这几个类别对应的 queue，示例代码如下(Swift 2)：

```
let qualityOfServiceClass = QOS_CLASS_BACKGROUND
let backgroundQueue = dispatch_get_global_queue(qualityOfServiceClass, 0)
dispatch_async(backgroundQueue, {
print("This is run on the background queue")
 dispatch_async(dispatch_get_main_queue(), { () -> Void in
     print("This is run on the main queue, after the previous code in outer b
lock")
  })
})
```

3. 主队列 主队列可以通过 `dispatch_get_main_queue()` 获取：

```
dispatch_async(dispatch_get_main_queue(), ^{
        // Update the UI
        [imageVIew setImage:image];
    });
```

自己创建的队列与系统队列有什么不同？

事实上，我们自己创建的队列，最终会把任务分配到系统提供的主队列和四个全局的并行队列上，这种操作叫做 Target queues。具体来说，我们创建的串行队列的 target queue 就是系统的主队列，我们创建的并行队列的 target queue 默认是系统 default 优先级的全局并行队列。所有放在我们创建的队列中的任务，最终都会到 target queue 中完成真正的执行。

那岂不是自己创建队列就没有什么意义了？其实不是的。通过我们自己创建的队列，以及 dispatch_set_target_queue 和 barrier 等操作，可以实现比较复杂的任务之间的同步，可以参考这里 和 这里。

通常情况下，对于串行队列，我们应该自己创建，对于并行队列，就直接使用系统提供的 Default 优先级的 queue。

注意：对于 `dispatch_barrier` 系列函数来说，传入的函数应当是自己创建的并行队列，否则 barrier 将失去作用。详情请参考苹果文档。

创建的 **Queue** 需要释放吗？

在 iOS6 之前，使用 `dispatch_queue_create` 创建的 queue 需要使用 `dispatch_retain` 和 `dispatch_release` 进行管理，在 iOS 6 系统把 dispatch queue 也纳入了 ARC 管理的范围，就不需要我们进行手动管理了。使用这两个函数会导致报错。

iOS6 上这个改变，把 dispatch queue 从原来的非 OC 对象（原生 C 指针），变成了 OC 对象，也带来了代码上的一些兼容性问题。在 iOS5 上需要使用 assign 来修饰 queue 对象：

```
@property (nonatomic, assign) dispatch_queue_t queue;
```

到 iOS6 以上就需要使用 strong 或者 weak 来修饰，不然会报错：

```
@property (nonatomic, strong) dispatch_queue_t queue;
```

当出现兼容性问题的时候，需要根据情况来修改代码，或者改变所 target 的 iOS 版本。

执行任务

折腾了半天 queue，现在终于到了让 queue 真正去执行任务的阶段了。给 queue 添加任务有两种方式，同步和异步。同步方式会阻塞当前线程的执行，等待添加的任务执行完毕之后，才继续向下执行。异步方式不会阻塞当前线程的执行。

```
dispatch_queue_t myCustomQueue;
myCustomQueue = dispatch_queue_create("com.example.MyCustomQueue", NULL);

// 异步添加
dispatch_async(myCustomQueue, ^{
    printf("做一些工作\n");
});

printf("第一个 block 可能还没有执行\n");

// 同步添加
dispatch_sync(myCustomQueue, ^{
    printf("做另外一些工作\n");
});
printf("两个 block 都已经执行完毕\n");
```

注意事项

- 同步和异步添加，与队列是串行队列和并行队列没有关系。可以同步地给并行队列添加任务，也可以异步地给串行队列添加任务。同步和异步添加只影响是不是阻塞当前线程，和任务的串行或并行执行没有关系
- 如果在任务 block 中创建了大量对象，可以考虑在 block 中添加 autorelease pool。尽管每个 queue 自身都会有 autorelease pool 来管理内存，但是 pool 进行 drain 的具体时间是没办法确定的。如果应用对于内存占用比较敏感，可以自己创建 autorelease pool 来进行内存管理。

关于线程安全

- Dispatch Queue 本身是线程安全的，换句话说，你可以从系统的任何一个线程给 queue 添加任务，不需要考虑加锁和同步问题
- 避免在任务中使用锁，如果使用锁的话可能会阻碍 queue 中其他 task 的运行
- 不建议获取 dispatch_queue 底层所使用的 thread 的有关信息，也不建议在 queue 中再使用 pthread 系函数

GCD 案例分析

案例一

这是一个广为流传的例子，代码如下：

```
NSLog(@"1"); // 任务1
dispatch_sync(dispatch_get_main_queue(), ^{
    NSLog(@"2"); // 任务2
});
NSLog(@"3"); // 任务3
```

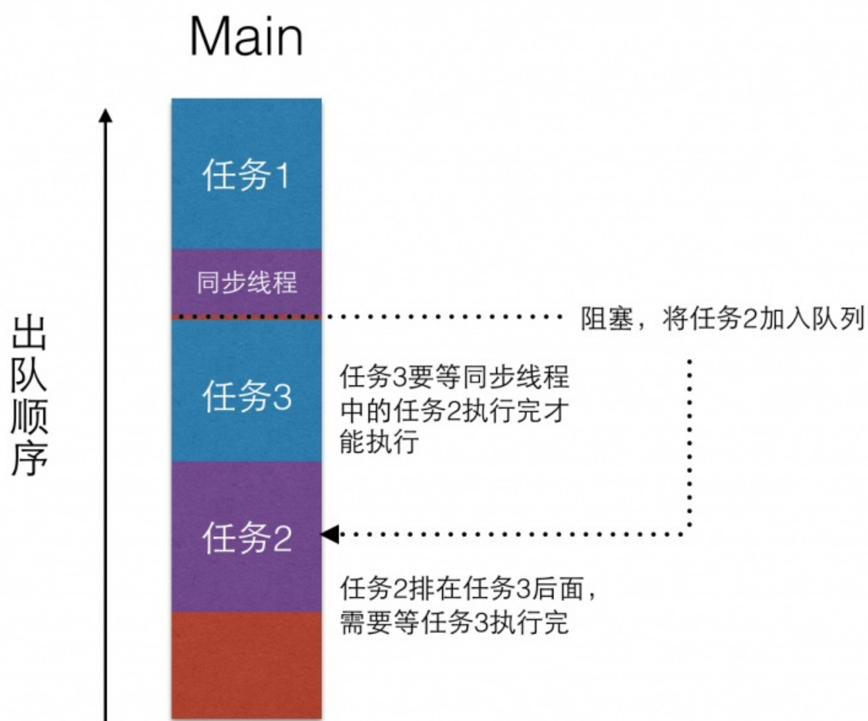
控制台输出

```
1
```

分析：

1. dispatch_sync 表示这是一个同步线程
2. dispatch_get_main_queue 表示其运行在主线程中的主队列
3. 任务2是同步线程的任务。

如图所示：



过程描述：

主线程启动以后的加入顺序是：任务1，同步线程，任务三。执行完任务1，就会启动同步线程，然后将任务2加入队列。所以，任务3在任务2的前面。如图中所示的那样，这种情况下任务2 与 任务 3都在等待彼此完成之后才能执行，这就造成了死锁。

案例二

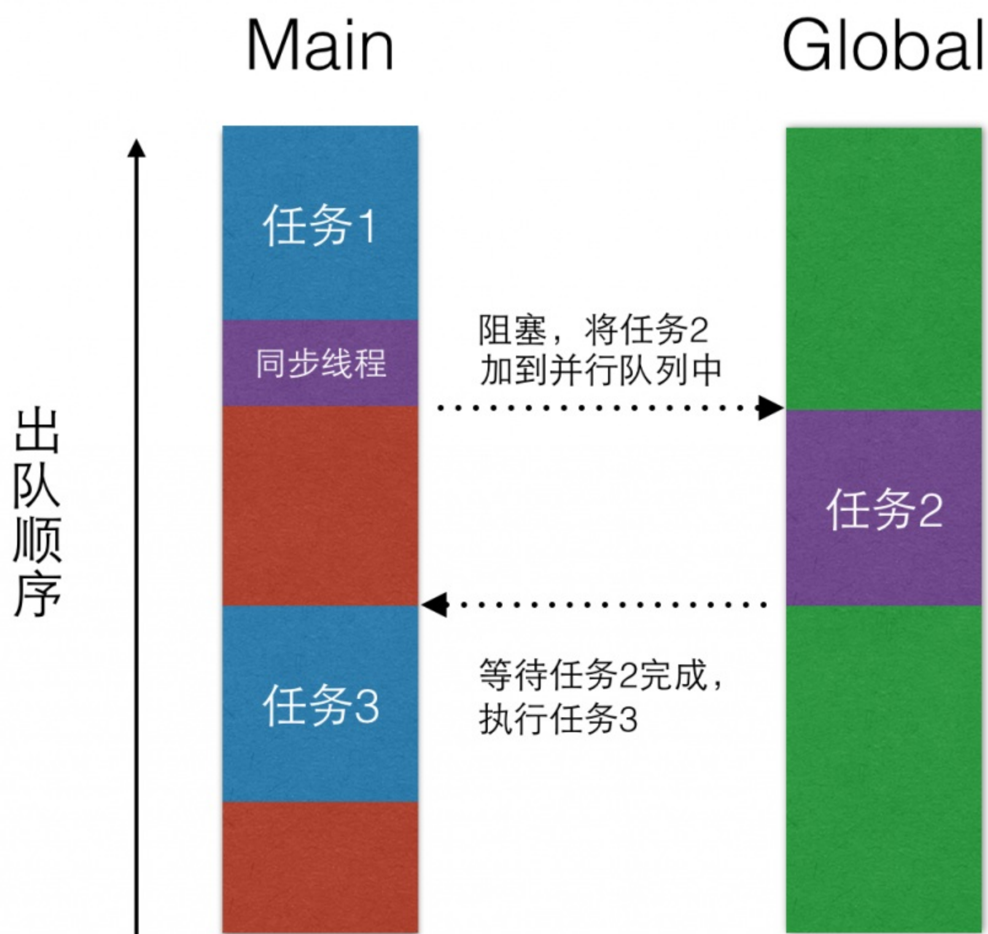
这个例子由此前的案例一演化而来，代码如下：

```
NSLog(@"1"); // 任务1
dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
    NSLog(@"2"); // 任务2
});
NSLog(@"3"); // 任务3
```

这并不会造成死锁，控制台输出如下：

```
1
2
3
```

如图所示：



分析与过程描述：

首先执行任务1，接下来会遇到一个同步线程，程序会进入等待。等待任务2执行完成以后，才能继续执行任务3。从 dispatch_get_global_queue 可以看出，任务2被加入到了全局的并行队列中，当并行队列执行完任务2以后，返回到主队列，继续执行任务3。

案例三

这个例子会比此前的两节复杂一些，代码如下：

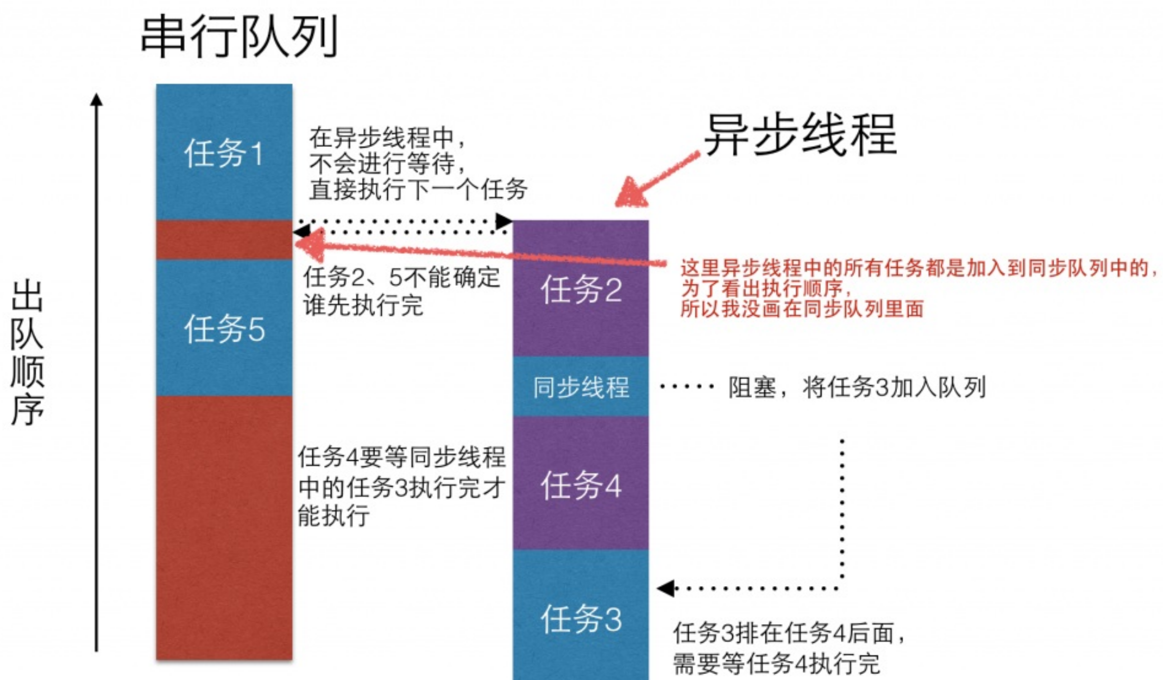
```
dispatch_queue_t queue = dispatch_queue_create("com.demo.serialQueue", DISPATCH_QUEUE_
SERIAL);
NSLog(@"1"); // 任务1
dispatch_async(queue, ^{
    NSLog(@"2"); // 任务2
    dispatch_sync(queue, ^{
        NSLog(@"3"); // 任务3
    });
    NSLog(@"4"); // 任务4
});
NSLog(@"5"); // 任务5
```

控制台输出如下：

```
1
5
2
// 5和2的顺序不一定
```

分析：这里没有使用系统提供的串行或并行队列，而是自己通过dispatch_queue_create函数创建了一个 `DISPATCH_QUEUE_SERIAL` 的串行队列。

如图所示：



过程描述：

1. 执行任务1
2. 遇到异步线程，将【任务2、同步线程、任务4】加入串行队列。因为是异步线程，所以在主线程中的任务5不必等待异步线程中的所有任务完成

3. 因为任务5不必等待，所以2和5的输出顺序不能确定
4. 任务2执行完以后，遇到同步线程，这时，将任务3加入异步的串行队列
5. 又因为任务4比任务3早加入串行队列，所以，任务3要等待任务4完成以后，才能执行。但是任务3所在的同步线程会阻塞，所以任务4必须等任务3执行完以后再执行。这就又陷入了无限的等待中，造成死锁。

案例四

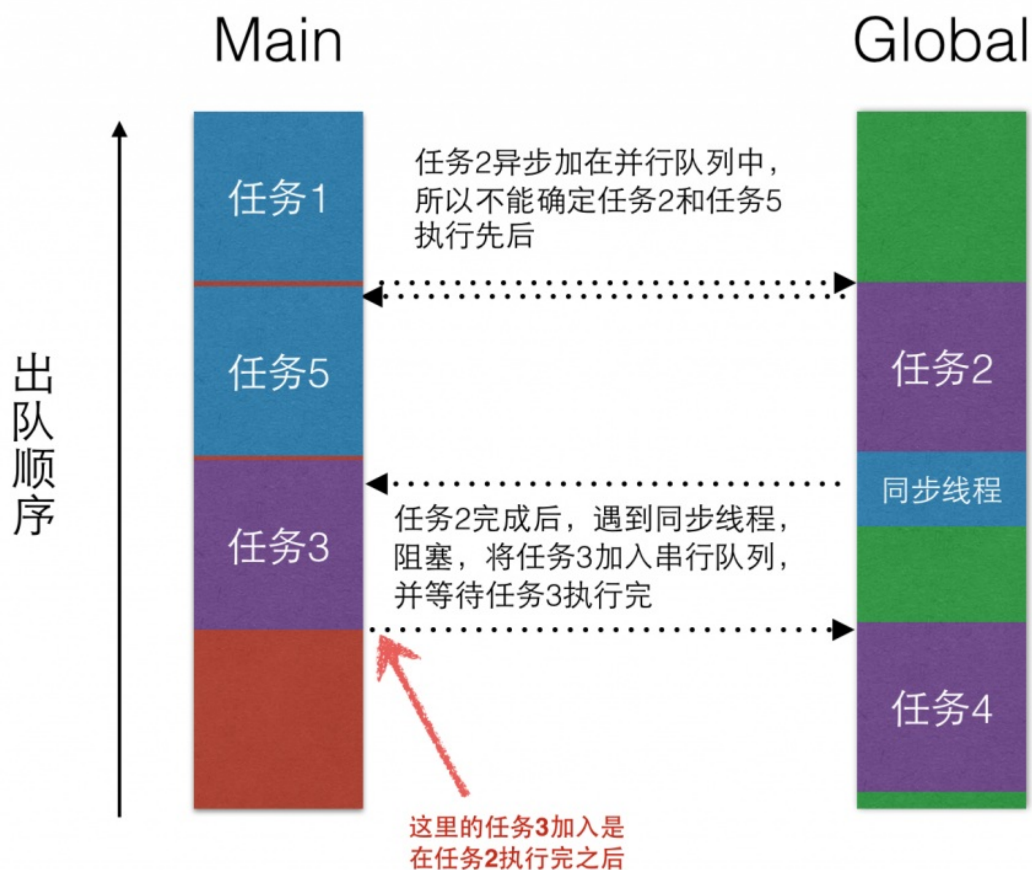
代码如下：

```
NSLog(@"1"); // 任务1
dispatch_async(dispatch_get_global_queue(0, 0), ^{
    NSLog(@"2"); // 任务2
    dispatch_sync(dispatch_get_main_queue(), ^{
        NSLog(@"3"); // 任务3
    });
    NSLog(@"4"); // 任务4
});
NSLog(@"5"); // 任务5
```

输出结果如下：

```
1
2
5
3
4
// 5和2的顺序不一定
```

如图所示：



分析与过程描述：

首先，将【任务1、异步线程、任务5】加入Main Queue中，异步线程中的任务是：【任务2、同步线程、任务4】。

所以，先执行任务1，然后将异步线程中的任务加入到Global Queue中，因为异步线程，所以任务5不用等待，结果就是2和5的输出顺序不一定。

然后再看异步线程中的任务执行顺序。任务2执行完以后，遇到同步线程。将同步线程中的任务加入到Main Queue中，这时加入的任务3在任务5的后面。

当任务3执行完以后，没有了阻塞，程序继续执行任务4。

从以上的分析来看，得到的几个结果：1最先执行；2和5顺序不一定；4一定在3后面。

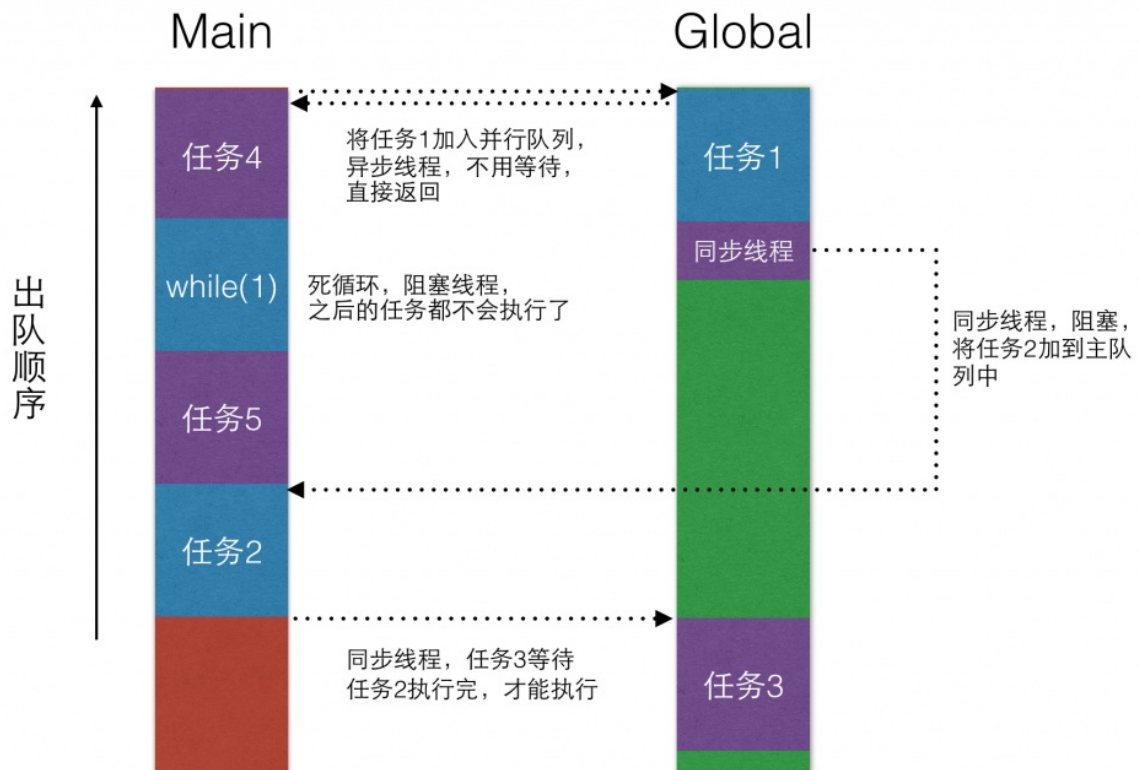
案例五

代码如下：

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{
    NSLog(@"1"); // 任务1
    dispatch_sync(dispatch_get_main_queue(), ^{
        NSLog(@"2"); // 任务2
    });
    NSLog(@"3"); // 任务3
});
NSLog(@"4"); // 任务4
while (1) {
}
NSLog(@"5"); // 任务5
```

输出如下：

```
1
4
// 1和4的顺序不一定
```



分析与过程描述：

和上面几个案例的分析类似，先来看看都有哪些任务加入了Main Queue：【异步线程、任务4、死循环、任务5】。

在加入到Global Queue异步线程中的任务有：【任务1、同步线程、任务3】。

第一个就是异步线程，任务4不用等待，所以结果任务1和任务4顺序不一定。

任务4完成后，程序进入死循环，Main Queue阻塞。但是加入到Global Queue的异步线程不受影响，继续执行任务1后面的同步线程。

同步线程中，将任务2加入到了主线程，并且，任务3等待任务2完成以后才能执行。这时的主线程，已经被死循环阻塞了。所以任务2无法执行，当然任务3也无法执行，在死循环后的任务5也不会执行。

最终，只能得到1和4顺序不定的结果。

案例总结

相信对于绝大多数人来说，在案例三开始，是否死锁以及整个的执行流程就变得不是那么显而易见了，这五个案例就意在展示 GCD 的问题：如果想要设置线程间的依赖关系，那就需要嵌套，如果嵌套就会导致一些复杂的事情发生。这应该是 GCD 的一个非常明显的缺陷之一了。

当然，NSOperation 为了我们提供了很方便设置依赖关系的解决方案。

NSOperation 和 NSOperationQueue

虽然标题这么写，但是实际上 NSOperation 和 NSOperationQueue 并不一定要一起使用。NSOperation 本身是可以单独使用的，不过单独使用的话并不能体现出 NSOperation 的强大之处（从下面的部分你就能看出单独用 NSOperation 真的是做不了什么事情），通常还是使用 NSOperationQueue 来执行 NSOperation。

NSOperation 是一个抽象类，我们需要继承它并且实现我们的子类。

并发和非并发

首先看一下不使用 OperationQueue 的情况。

默认情况下 NSOperation 是非并发的，当我们像下面这样定义一个 operation:

```
@implementation MyOperation

-(void)main {
    NSLog(@"MyOperation Main Function");
}

@end
```

然后启动它：

```
#import "MyOperation.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        MyOperation *op = [[MyOperation alloc] init];
        [op start];
        NSLog(@"Main Function");
    }
    return 0;
}
```

可以看到运行结果是：

```
MyOperation Main Function
Main Function
```

即整个 Operation 就是在当前的线程中以阻塞的形式执行的，当 operation 的 main 函数执行完毕之后，程序的控制权返回到主的 main 函数中。这样看来 operation 跟普通的一个函数调用就没有什么区别了。

对于并发的 Operation，要实现还是有点麻烦的，我们需要重载 start，isAsynchronous，isExecuting，isFinished 四个函数，同时还最好在 start 和 main 的实现中支持 cancel 操作。为什么要这么麻烦呢？因为对于一个并发的 Operation，调用者知道它什么时候开始，却不能知道它什么时候结束。在 NSOperation 的体系下，是通过 KVO 监测 isExecuting 和 isFinished 这几个变量，来监测 Operation 的完成状态的。出于兼容性的考虑（参考这里），我们还必须手动触发 KVO 通知。下面是一个示例：

```
#import "MyOperation.h"

@interface MyOperation()

@property (atomic, assign) BOOL _executing;
@property (atomic, assign) BOOL _finished;
@end

@implementation MyOperation

- (void)start;
{
    if ([self isCancelled])
    {
        // Move the operation to the finished state if it is canceled.
        [self willChangeValueForKey:@"isFinished"];
        self._finished = YES;
        [self didChangeValueForKey:@"isFinished"];
        return;
    }

    // If the operation is not canceled, begin executing the task.
    [self willChangeValueForKey:@"isExecuting"];
    [NSThread detachNewThreadSelector:@selector(main) toTarget:self withObject:nil];
    self._executing = YES;
    [self didChangeValueForKey:@"isExecuting"];

}

- (void)main;
{
    if ([self isCancelled]) {
        return;
    }
    sleep(10);
    NSLog(@"MyOperation Main Function");
    [self completeOperation];
}

- (BOOL)isAsynchronous;
{
    return YES;
}

- (BOOL)isExecuting {
    return self._executing;
}

- (BOOL)isFinished {
    return self._finished;
}

- (void)completeOperation {
    [self willChangeValueForKey:@"isFinished"];
    [self willChangeValueForKey:@"isExecuting"];

    self._executing = NO;
    self._finished = YES;

    [self didChangeValueForKey:@"isExecuting"];
    [self didChangeValueForKey:@"isFinished"];
}
@end
```

可以看到所谓的“并发”，跟上面的非并发并没有什么本质的不同，完全取决于我们的 start 函数是如何实现的。这里我们的 start 函数中把任务直接扔给了另外的线程，也就不会阻塞当前线程了。

废了这么大劲，我们如何执行这个 Operation 呢？如果再像上面一样使用 `[op start]` 直接执行的话，你会发现还没等到 Operation 返回我们的整个程序就已经结束掉了。因为我们的主程序并不会等到 operatoin 返回。想要等到 operation 返回，我们还需要手动地去监视 operation 的变量，然后等待它返回。。。

看到这里你就明白为什么单独使用 NSOperation 发挥不了太大的作用了，因为 NSOperation 本身确实是没有做什么工作，大部分东西还是要靠我们自己来控制。

这时候就需要 NSOperationQueue 登场了。

在 NSOperationQueue 中运行

NSOperationQueue 是一个专门用于执行 NSOperation 的队列。在 OS X 10.6 之后，把一个 NSOperation 放到 NSOperationQueue 中，queue 会忽略 isAsynchronous 变量，总是会把 operation 放到后台线程中执行。这样不管 operation 是不是异步的，queue 的执行都是不会造成主线程的阻塞的。使用 Queue 可以很方便地进行并发操作，并且帮我们完成大部分的监视 operation 是否完成的操作。接着用上面的 MyOperation 做例子，使用 NSOperationQueue 之后，我们就可以这样写：

```
MyOperation *op = [[MyOperation alloc] init];
NSOperationQueue *queue = [[NSOperationQueue alloc] init];

[queue addOperation:op];     // add 完 operation 就立即启动了
[queue waitUntilAllOperationsAreFinished]; // 阻塞当前线程，直到所有的 operation 全都完成
NSLog(@"Main Function");
```

像这样，我们可以添加各个各样的 operation 到 queue 中，只要这些 operation 都正确地重载了 isExecuting 和 isFinished，就可以正确地被并发执行。

除此之外，NSOperationQueue 还有几个很强大的特性。

Dependency

NSOperation 可以通过 addDependency 来依赖于其他的 operation 完成，如果有很多复杂的 operation，我们可以形成它们之间的依赖关系图，来实现复杂的同步操作：

```
[updateUIOperation addDependency: workerOperation];
```

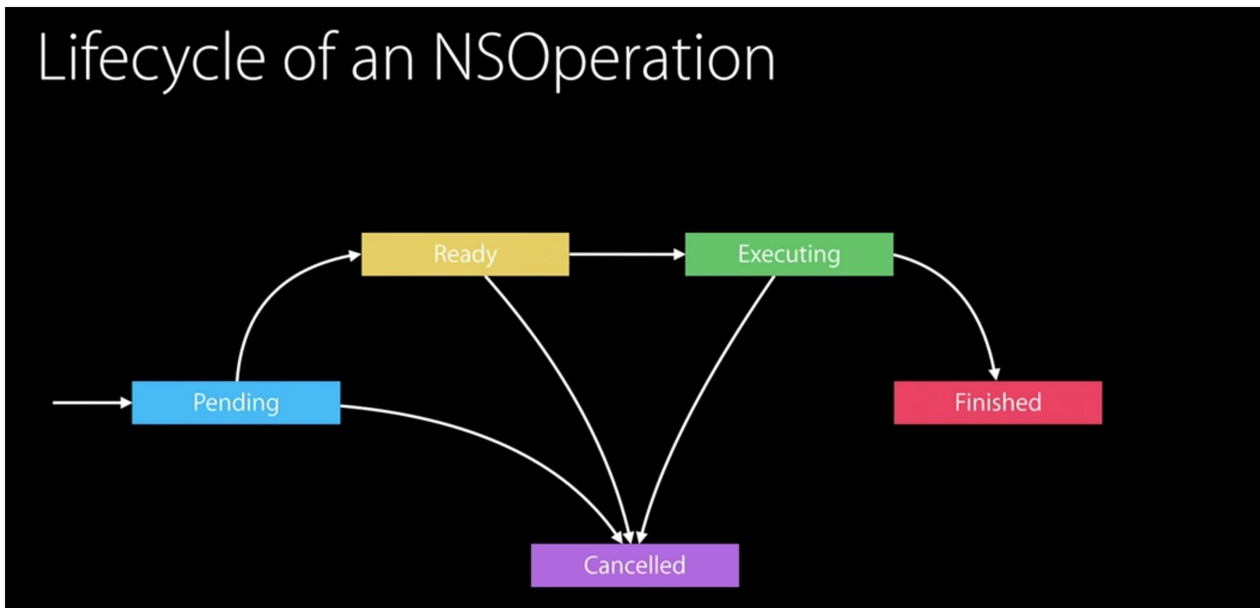
Cancellation

NSOperation 有如下几种的运行状态：

- Pending
- Ready
- Executing
- Finished

- Canceled

除 Finished 状态外，其他状态均可转换为 Canceled 状态。



当 NSOperation 支持了 cancel 操作时，NSOperationQueue 可以使用 cancelAllOperatoins 来对所有的 operation 执行 cancel 操作。不过 cancel 的效果还是取决于 NSOperation 中代码是怎么写的。比如 对于数据库的某些操作线程来说，cancel 可能会意味着 你需要把数据恢复到最原始的状态。

maxConcurrentOperationCount

默认的最大并发 operation 数量是由系统当前的运行情况决定的(来源)，我们也可以强制指定一个固定的并发数量。

Queue 的优先级

NSOperationQueue 可以使用 queuePriority 属性设置优先级，具体的优先级有下面几种：

```
typedef enum : NSInteger {
   NSOperationQueuePriorityVeryLow = -8,
   NSOperationQueuePriorityLow = -4,
   NSOperationQueuePriorityNormal = 0,
   NSOperationQueuePriorityHigh = 4,
   NSOperationQueuePriorityVeryHigh = 8
} NSOperationQueuePriority;
```

在 Queue 中优先级较高的会先执行。

注1：尽管系统会尽量使得优先级高的任务优先执行，不过并不能确保优先级高的任务一定会先于优先级低的任务执行，即优先级并不能保证任务的执行先后顺序。要先让一个任务先于另一个任务执行，需要使用设置dependency 来实现。

注**2**：同 NSOperation 一样，NSOperationQueue 也具有若干 QoS 选项可供选择。有关 QoS 配置的具体细节，例如当 NSOperation 和 NSOperationQueue 具有不同的 QoS 时出现的效果，以及如何改变 QoS 等，可以参考苹果官方文档 Energy Efficiency Guide for iOS Apps 。

GCD 与 NSOperation 的对比

这是面试中经常会问到的一点，这两个都很常用，也都很强大。对比它们可以从下面几个角度来说：

- 首先要明确一点，NSOperationQueue 是基于 GCD 的更高层的封装，从 OS X 10.10 开始可以通过设置 `underlyingQueue` 来把 operation 放到已有的 dispatch queue 中。
- 从易用性角度，GCD 由于采用 C 风格的 API，在调用上比使用面向对象风格的 NSOperation 要简单一些。
- 从对任务的控制性来说，NSOperation 显著得好于 GCD，和 GCD 相比支持了 Cancel 操作（注：在 iOS8 中 GCD 引入了 `dispatch_block_cancel` 和 `dispatch_block_testcancel` ，也可以支持 Cancel 操作了），支持任务之间的依赖关系，支持同一个队列中任务的优先级设置，同时还可以通过 KVO 来监控任务的执行情况。这些通过 GCD 也可以实现，不过需要很多代码，使用 NSOperation 显得方便了很多。
- 从第三方库的角度，知名的第三方库如 AFNetworking 和 SDWebImage 背后都是使用 NSOperation，也从另一方面说明对于需要复杂并发控制的需求，NSOperation 是更好的选择（当然也不是绝对的，例如知名的 Parse SDK 就完全没有使用 NSOperation，全部使用 GCD，其中涉及到大量的 GCD 高级用法，这里有相关解析）。

参考资料

- http://www.raywenderlich.com/19788/how-to-use-nsoperations-and-nsoperationqueues
- http://www.humancode.us/2014/08/14/target-queues.html
- http://www.dribin.org/dave/blog/archives/2009/05/05/concurrent_operations/
- http://www.jianshu.com/p/0b0d9b1f1f19
- http://www.cnblogs.com/tangbinblog/p/4133481.html
- http://www.saitjr.com/ios/ios-gcd-deadlock.html

单例模式（Singleton）

单例模式是一种常见的设计模式，在 Cocoa 开发中也经常使用。

一个简单的单例模式示例代码如下：

```
/* Singleton.h */
#import "Foundation/Foundation.h"
@interface Singleton : NSObject
+ (Singleton *)shardInstance;
@end

/* Singleton.m */
#import "Singleton.h"
static Singleton *instance = nil;

@implementation Singleton
+ (Singleton *)sharedInstance {
    if (!instance) {
        instance = [[super allocWithZone:NULL] init];
    }
    return instance;
}
```

Cocoa 库本身在一些地方也使用了单例模式，例如 `[NSNotificationCenter defaultCenter]` ， `[UIColor redColor]` 等。

这种写法的优点是，可以延迟加载，按需分配内存以节省开销。

但是，这并非一个线程安全的写法，比如两个或多个线程并发的调用 `sharedInstance` 方法，有可能会得到多个实例，这里列出两种方法来创建一个线程安全的单例。

@synchronized

可以使用@synchronized进行加锁，代码如下：

```
/* Singleton.h */
#import <Foundation/Foundation.h>
@interface Singleton : NSObject
+ (Singleton *)sharedInstance;
@end
/* Singleton.m */
#import "Singleton.h"
static Singleton *instance = nil;
@implementation Singleton
+ (Singleton *)sharedInstance {
    @synchronized (self) {
        if (!instance) {
            instance = [[super alloc] init];
        }
    }
    return instance;
}
```

这种写法也是懒加载，不过虽然保证了线程安全但是由于锁的存在当多线程访问时，性能会降低。

GCD

这里主要利用GCD中的dispatch_once方法，这是最普遍也是苹果最推荐的方法，函数原型如下：

```
void dispatch_once(
   dispatch_once_t *predicate,
   dispatch_block_t block);
```

单例实现代码如下：

```
/* Singleton.h */
#import <Foundation/Foundation.h>
@interface Singleton : NSObject
+ (Singleton *)sharedInstance;
@end
/* Singleton.m */
#import "Singleton.h"
static Singleton *instance = nil;
@implementation Singleton
+ (Singleton *)sharedInstance {
    static dispatch_once_t predicate;
    dispatch_once(&predicate, ^{
        instance = [[Singleton alloc] init];
    });
    return instance;
}
```

这样的方法有很多优势，首先满足了线程安全问题，其次很好满足静态分析器要求。

GCD 可以确保以更快的方式完成这些检测，它可以保证 block 中的代码在任何线程通过 dispatch_once 调用之前被执行，但它不会强制每次调用这个函数都让代码进行同步控制。

苹果的文档 documentation for dispatch_once 是这么说的：

> The predicate must point to a variable stored in global or static scope. The result of using a predicate with automatic or dynamic storage (including Objective-C instance variables) is undefined.

所以，如果你的 predicate 不是静态的、不是全局的，还是不能用GCD。其实如果去看这个函数所在的头文件，你会发现目前它的实现其实是一个宏。

工厂模式（Factory）

工厂模式是另一种常见的设计模式，本质上是使用方法来简化类的选择和初始化过程。

下面是一个网上到处都是的简单工厂模式的例子：

```
//
//  OperationFactory.m
//  FactoryPattern

#import "OperationFactory.h"
#import "Operation.h"
#import "OperationAdd.h"
#import "OperationSub.h"
#import "OperationMul.h"
#import "OperationDiv.h"

@implementation OperationFactory

+ (Operation *) createOperat:(char)operate{
    Operation *oper = nil;
    switch (operate) {
        case '+':
        {
            oper = [[OperationAdd alloc] init];
            break;
        }
        case '-':
        {
            oper = [[OperationSub alloc] init];
            break;
        }
        case '*':
        {
            oper = [[OperationMul alloc] init];
            break;
        }
        case '/':
        {
            oper = [[OperationDiv alloc] init];
            break;
        }
        default:
            break;
    }
    return oper;
}
@end
```

由于 Objective-C 本身的动态特性，还可以用反射来改写：

```
@implementation OperationFactory
+ (Operation *) createOperat:(NSString *)operate{
    Operation *oper = nil;
    Class class = NSClassFromString(operate);
    oper = [(Operation *)[class alloc] init];
    if ([oper respondsToSelector:@selector(getResult)]) {
        [oper getResult];
    }
    return oper;
}
@end
```

使用时，可以传入类名，来获取对应类的对象：

```
Operation *oper = [OperationFactory createOperat: @"OperationAdd"];
oper.numberA = 10;
oper.numberB = 20;
NSLog(@"%f", oper.getResult);
```

委托模式（Delegate）

委托模式是 Cocoa 中十分常见的设计模式，在 Cocoa 库中被大量的使用。在 Objective-C 中，委托模式通常使用协议（protocol）来实现。

委托模式的示例代码：

```
@protocol PrintDelegate <NSObject>
- (void)print;
@end


@interface AClass : NSObject<PrintDelegate>
@property id<PrintDelegate> delegate;
@end

@implementation AClass

-(void)sayHello {
    [self.delegate print];
}

-(void)print {
    NSLog(@"Do Print");
}
@end

// 使用 AClass
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        AClass * a = [AClass new];
        a.delegate = a;
        [a sayHello];
    }
    return 0;
}
```

这里对象a的 delegate 设置为自己，也可以是任何一个实现了 `PrintDelegate` 协议的对象。

观察者模式（Observer）

Cocoa 中提供了两种用于实现观察者模式的办法，一直是使用 `NSNotification`，另一种是 `KVO(Key Value Observing)`。

NSNotification

`NSNotification` 基于 Cocoa 自己的消息中心组件 `NSNotificationCenter` 实现。

观察者需要统一在消息中心注册，说明自己要观察哪些值的变化。观察者通过类似下面的函数来进行注册：

```
[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(printName:)
                                             name: @"messageName"
                                           object:nil];
```

上面的函数表明把自身注册成 "messageName" 消息的观察者，当有消息时，会调用自己的 `printName` 方法。

消息发送者使用类似下面的函数发送消息：

```
[[NSNotificationCenter defaultCenter] postNotificationName:@"messageName"
                                                    object:nil
                                                  userInfo:nil];
```

KVO(Key Value Observing)

KVO的实现依赖于 Objective-C 本身强大的 KVC(Key Value Coding) 特性，可以实现对于某个属性变化的动态监测。

示例代码如下：

```
// Book类
@interface Book : NSObject

@property NSString *name;
@property CGFloat price;

@end

// AClass类
@class Book;
@interface AClass : NSObject

@property (strong) Book *book;

@end

@implementation AClass

- (id)init:(Book *)theBook {
    if(self = [super init]){
        self.book = theBook;
        [self.book addObserver:self forKeyPath:@"price" options:NSKeyValueObservingOpt
ionOld|NSKeyValueObservingOptionNew context:nil];
    }
    return self;
}

- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context{
    if([keyPath isEqual:@"price"]){
        NSLog(@"------price is changed------");
        NSLog(@"old price is %@",[change objectForKey:@"old"]);
        NSLog(@"new price is %@",[change objectForKey:@"new"]);
    }
}

- (void)dealloc{
    [self.book removeObserver:self forKeyPath:@"price"];
}
@end

// 使用 KVO
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Book *aBook = [Book new];
        aBook.price = 10.9;
        AClass * a = [[AClass alloc] init:aBook];
        aBook.price = 11; // 输出 price is changed
    }
    return 0;
}
```

参考资料

- Objective-C中的单例模式
- iPhone开发笔记——简单工厂模式
- 详解Objective-C中的委托和协议
- Objective-C中Observer模式的实现
- Objective-C KVO编程
- iOS开发系列——Objective-C开发之KVC，KVO

离屏渲染

离屏渲染往往会带来界面卡顿的问题，这里将会讨论 当前屏幕渲染、离屏渲染 以及 CPU 渲染

在 OpenGL 中，GPU 屏幕渲染有以下两种方式：

- On-Screen Rendering

即当前屏幕渲染，在用于显示的屏幕缓冲区中进行，不需要额外创建新的缓存，也不需要开启新的上下文，所以性能较好，但是受到缓存大小限制等因素，一些复杂的操作无法完成。

- Off-Screen Rendering

即离屏渲染，指的是在 GPU 的当前屏幕缓冲区外开辟新的缓冲区进行操作。

相比于当前屏幕渲染，离屏渲染的代价是很高的，主要体现在如下两个方面：

- 创建新的缓冲区
- 上下文切换。离屏渲染的整个过程，需要多次切换上下文环境：先从当前屏幕切换到离屏，等待离屏渲染结束后，将离屏缓冲区的渲染结果显示到到屏幕上，这又需要将上下文环境从离屏切换到当前屏幕。

当设置了以下属性时，会触发离屏渲染：

- shouldRasterize（光栅化）
- masks（遮罩）
- shadows（阴影）
- edge antialiasing（抗锯齿）
- group opacity（不透明）

为了避免卡顿问题，应当尽可能使用当前屏幕渲染，可以不使用离屏渲染则尽量不用，应当尽量避免使用 layer 的 border、corner、shadow、mask 等技术。必须离屏渲染时，相对简单的视图应该使用 CPU 渲染，相对复杂的视图则使用一般的离屏渲染。

如下是 CPU 渲染和离屏渲染的区别：

由于GPU的浮点运算能力比CPU强，CPU渲染的效率可能不如离屏渲染。但如果仅仅是实现一个简单的效果，直接使用 CPU 渲染的效率又可能比离屏渲染好，毕竟普通的离屏渲染要涉及到缓冲区创建和上下文切换等耗时操作。对一些简单的绘制过程来说，这个过程有可能用CoreGraphics，全部用CPU来完成反而会比GPU做得更好。一个常见的 CPU 渲染的例子是：重写 `drawRect` 方法，并且使用任何 Core Graphics 的技术进行了绘制操作，就涉及到了 CPU 渲染。整个渲染过程由 CPU 在 App 内同步地完成，渲染得到的 `bitmap` 最后再交由GPU用于显示。总之，具体使用 CPU 渲染还是使用 GPU 离屏渲染更多的时候需要进行性能上的具体比较才可以。

一个常见的性能优化的例子就是如何给 UIView/UIImageView 加圆角。

如下是三种加圆角的方式：

- 设置 cornerRadius
- UIBezierPath
- Core Graphics(为 UIView 加圆角)与直接截取图片(为 UIImageView 加圆角)

如下是这三种方法的比较：

cornerRadius

```
view.layer.cornerRadius = 6.0;
view.layer.masksToBounds = YES;
```

这种方式会触发两次离屏渲染，如果在滚动页面中这么做的话就会遇到性能问题。当然我们可以进行缓存以优化性能，如下：

```
view.layer.shouldRasterize = YES;
view.layer.rasterizationScale = [UIScreen mainScreen].scale;
```

shouldRasterize = YES 会使视图渲染内容被缓存起来，下次绘制的时候可以直接显示缓存，当然要在视图内容不改变的情况下。

注意：png 图片 在 UIImageView 这样处理圆角是不会产生离屏渲染的。（ios9.0之后不会离屏渲染，ios9.0之前还是会离屏渲染）。

UIBezierPath

```
- (void)drawRect:(CGRect)rect {
  CGRect bounds = self.bounds;
  [[UIBezierPath bezierPathWithRoundedRect:rect cornerRadius:8.0] addClip];

  [self.image drawInRect:bounds];
}
```

这种方法会触发一次离屏渲染，很多资料推崇这种写法，但是这种方式会导致内存暴增，并且同样会触发离屏渲染。

Core Graphics(为 UIView 加圆角)与直接截取图片(为 UIImageView 加圆角)

正如你所期待的那样，这种方法应该是极具效率的正确的姿势。这里将为 UIView 添加圆角与为 UIImageView 添加圆角进行区分。

使用 Core Graphics 为 UIView 加圆角

这种做法的原理是利用 Core Graphics 自己画出了一个圆角矩形。

```
func kt_drawRectWithRoundedCorner(radius radius: CGFloat,
                                  borderWidth: CGFloat,
                                  backgroundColor: UIColor,
                                  borderColor: UIColor) -> UIImage {
    UIGraphicsBeginImageContextWithOptions(sizeToFit, false, UIScreen.mainScreen().sca
le)
    let context = UIGraphicsGetCurrentContext()

    CGContextMoveToPoint(context, 开始位置);  // 开始坐标右边开始
    CGContextAddArcToPoint(context, x1, y1, x2, y2, radius);  // 这种类型的代码重复四次

    CGContextDrawPath(UIGraphicsGetCurrentContext(), .FillStroke)
    let output = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    return output
}
```

这个方法返回的是 UIImage，有了这个图片后，就可以创建一个 UIImageView 并插入到视图层级的底部：

```
extension UIView {
    func kt_addCorner(radius radius: CGFloat,
                      borderWidth: CGFloat,
                      backgroundColor: UIColor,
                      borderColor: UIColor) {
        let imageView = UIImageView(image: kt_drawRectWithRoundedCorner(radius: radius
,
                                    borderWidth: borderWidth,
                                    backgroundColor: backgroundColor,
                                    borderColor: borderColor))
        self.insertSubview(imageView, atIndex: 0)
    }
}
```

在调用时 只需要像这样写：

```
let view = UIView(frame: CGRectMake(1,2,3,4))
view.kt_addCorner(radius: 6)
```

直接截取图片为 UIImageView 加圆角

这里的实现思路是直接截取图片：

```
extension UIImage {
    func kt_drawRectWithRoundedCorner(radius radius: CGFloat, _ sizetoFit: CGSize) ->
UIImage {
        let rect = CGRect(origin: CGPoint(x: 0, y: 0), size: sizetoFit)

        UIGraphicsBeginImageContextWithOptions(rect.size, false, UIScreen.mainScreen()
.scale)
        CGContextAddPath(UIGraphicsGetCurrentContext(),
            UIBezierPath(roundedRect: rect, byRoundingCorners: UIRectCorner.AllCorners
,
                cornerRadii: CGSize(width: radius, height: radius)).CGPath)
        CGContextClip(UIGraphicsGetCurrentContext())

        self.drawInRect(rect)
        CGContextDrawPath(UIGraphicsGetCurrentContext(), .FillStroke)
        let output = UIGraphicsGetImageFromCurrentImageContext();
        UIGraphicsEndImageContext();

        return output
    }
}
```

圆角路径直接用贝塞尔曲线绘制。这个函数的效果是将原来的 UIImage 剪裁出圆角。配合着这函数，我们可以为 UIImageView 拓展一个设置圆角的方法：

```
extension UIImageView {
    /**
     / !!!只有当 imageView 不为nil 时，调用此方法才有效果

     :param: radius 圆角半径
     */
    override func kt_addCorner(radius radius: CGFloat) {
        self.image = self.image?.kt_drawRectWithRoundedCorner(radius: radius, self.bou
nds.size)
    }
}
```

在调用时只需要像如下这样写：

```
let imageView = let imgView1 = UIImageView(image: UIImage(name: ""))
imageView.kt_addCorner(radius: 6)
```

> 注意：需要小心使用背景颜色。因为没有设置 `masksToBounds`，因此超出圆角的部分依然会被显示。因此不应该再使用背景颜色，可以在绘制圆角矩形时设置填充颜色来达到类似效果。

总结

- 如果能够只用 cornerRadius 解决问题，就不用优化。
- 如果必须设置 masksToBounds，可以参考圆角视图的数量，如果数量较少（一页只有几个）也可以考虑不用优化。
- UIImageView 的圆角通过直接截取图片实现，其它视图的圆角可以通过 Core Graphics 画出圆角矩形实现。

参考链接

- 小心别让圆角成了你列表的帧数杀手
- http://blog.ibireme.com/2015/11/12/smooth_user_interfaces_for_ios/
- https://medium.com/ios-os-x-development/perfect-smooth-scrolling-in-uitableviews-fd609d5275a5
- UIKit性能调优
- http://articles.cocoahope.com/blog/2013/03/06/applying-rounded-corners

Swift

本以为 Swift 3 之后 Swift 就会稳定了，听闻 Swift 4 还会有 breaking change，本文档 Swift 部分暂时先搁置吧...

# #

- https://github.com/ChenYilong/iOSInterviewQuestions
- http://draveness.me/guan-yu-xie-ios-wen-ti-de-jie-da/
- https://github.com/lzyy/iOS-Developer-Interview-Questions

更多学习资料：

- http://nshipster.cn
- http://objccn.io/
- http://arigrant.com/
- http://oncenote.com/
- https://github.com/100mango/zen
- https://github.com/oa414/objc-zen-book-cn
- https://github.com/nixzhu/dev-blog
- https://github.com/robovm/apple-ios-samples （苹果官方 Sample 代码集合）
- https://github.com/leecade/ios-dev-flow （开发流程总结）
- https://github.com/DaiYue/iOS-good-practices-in-Chinese （iOS 最佳实践）
- https://github.com/tangqiaoboy/iOSBlogCN （iOS 开发博客列表）
- https://github.com/Aufree/trip-to-iOS （iOS 学习资料整理）
- http://www.hrchen.com/2013/05/performance-with-instruments/ (iOS 性能优化）
- https://github.com/huang303513/iOS-Study-Demo
- https://github.com/seedante/iOS-Note
- https://zsisme.gitbooks.io/ios-/content/index.html（iOS Core Animation: Advanced Techniques中文译本）

编码规范

- https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html#//apple_ref/doc/uid/10000146-SW1
- https://github.com/raywenderlich/objective-c-style-guide
- https://github.com/github/objective-c-style-guide

常用的库总结：

- http://github.ibireme.com/github/list/ios/
- https://github.com/Tim9Liu9/TimLiu-iOS
- https://github.com/vsouza/awesome-ios
- https://github.com/cjwirth/awesome-ios-ui

1. 垃圾回收机制。。。(主要从下面几方面解答 GC原理、最好画图解释一下年轻代（Eden区和Survival区）、年老代、比例分配及为啥要这样分代回收)
2. 对象分配问题，堆栈里的问题，详细的会问道方法区、堆、程序计数器、本地方法栈、虚拟机栈，问题入口从String a,new String("")开始
3. 关键字，private protected public static final 组合着问
4. Object类里面有哪几种方法，作用
5. equals 和 hashCode方法，重写equals的原则()
6. 向上转型
7. Java引用类型(强引用，软引用，弱引用，虚引用)
8. 线程相关的，主要是volitate，synchorized，wait()，notify()，notifyAll()，join()
9. Exception和Error
10. 反射的用途
11. HashMap实现原理(数组+链表)，查找数据的时间复杂度
12. List有哪些子类，各有什么区别
13. NIO相关，缓冲区、通道、selector。。。(不熟，面了这么多，挂在这里。其实主要是表现在同步阻塞和异步，传输方式不同。标准IO无法实现非阻塞模式、文件锁、读选择、分散聚集等)
14. 内存泄露，举个例子
15. OOM是怎么出现的，有哪几块JVM区域会产生OOM，如何解决(对于该问题，建议去《Java特种兵》的3.6章)
16. Java里面的观察者模式实现
17. 单例实现(我一般用enum写，不容易被挑毛病)
18. 用Java模拟一个栈，并能够做到扩容，并且能有同步锁。（用数组实现）
19. Java泛型机制，泛型机制的优点，以及类型变量

总的来说，Android的系统体系结构分为四层，自顶向下分别是：

- 应用程序(Applications)
- 应用程序框架(Application Frameworks)
- 系统运行库与Android运行环境(Libraris & Android Runtime)
- Linux内核(Linux Kernel)

安卓系统结构示意图



下面对每层进行详细说明

1. 应用程序(Applications)

Android会同一系列核心应用程序包一起发布，该应用程序包包括email客户端，SMS短消息程序，日历，地图，浏览器，联系人管理程序等。所有的应用程序都是使用JAVA语言编写的。通常开发人员就处在这一层。

2. 应用程序框架(Application Frameworks)

提供应用程序开发的各种API进行快速开发，也即隐藏在每个应用后面的是一系列的服务和系统，大部分使用Java编写，所谓官方源码很多也就是看这里，其中包括：

- 丰富而又可扩展的视图（Views），可以用来构建应用程序， 它包括列表（lists），网格（grids），文本框（text boxes），按钮（buttons）， 甚至可嵌入的web浏览器。

- 内容提供器（Content Providers）使得应用程序可以访问另一个应用程序的数据（如联系人数据库），或者共享它们自己的数据
- 资源管理器（Resource Manager）提供 非代码资源的访问，如本地字符串，图形，和布局文件（ layout files ）。
- 通知管理器 （Notification Manager） 使得应用程序可以在状态栏中显示自定义的提示信息。
- 活动管理器（ Activity Manager） 用来管理应用程序生命周期并提供常用的导航回退功能。

3. 系统运行库与Android运行环境(Libraris & Android Runtime)

1) 系统运行库

Android 包含一些C/C++库，这些库能被Android系统中不同的组件使用。它们通过 Android 应用程序框架为开发者提供服务。以下是一些核心库：

- **Bionic**系统 **C** 库 - 一个从 BSD 继承来的标准 C 系统函数库（ libc ）， 它是专门为基于 embedded linux 的设备定制的。
- 媒体库 - 基于 PacketVideo OpenCORE；该库支持多种常用的音频、视频格式回放和录制，同时支持静态图像文件。编码格式包括MPEG4, H.264, MP3, AAC, AMR, JPG, PNG 。
- **Surface Manager** - 对显示子系统的管理，并且为多个应用程序提 供了2D和3D图层的无缝融合。这部分代码
- **Webkit,LibWebCore** - 一个最新的web浏览器引擎用，支持Android浏览器和一个可嵌入的web视图。鼎鼎大名的 Apple Safari背后的引擎就是Webkit
- **SGL** - 底层的2D图形引擎
- **3D libraries** - 基于OpenGL ES 1.0 APIs实现；该库可以使用硬件 3D加速（如果可用）或者使用高度优化的3D软加速。
- **FreeType** -位图（bitmap）和矢量（vector）字体显示。
- **SQLite** - 一个对于所有应用程序可用，功能强劲的轻型关系型数据库引擎。
- 还有部分上面没有显示出来的就是硬件抽象层。其实Android并非讲所有的设备驱动都放在linux内核里面，而是实现在userspace空间，这么做的主要原因是GPL协议，Linux是遵循该 协议来发布的，也就意味着对 linux内核的任何修改，都必须发布其源代码。而现在这么做就可以避开而无需发布其源代码，毕竟它是用来赚钱的。 而 在linux内核中为这些userspace驱动代码开一个后门，就可以让本来userspace驱动不可以直接控制的硬件可以被访问。而只需要公布这个 后门代码即可。一般情况下如果要将Android移植到其他硬件去运行，只需要实现这部分代码即可。包括：显示器驱动，声音，相机，GPS,GSM等等

2) Android运行环境

该核心库提供了JAVA编程语言核心库的大多数功能。
每一个Android应用程序都在它自己的进程中运 行，都拥有一个独立的Dalvik虚拟 机实例。Dalvik被设计成一个设备可以同时高效地运行多个虚拟系统。 Dalvik虚拟机执行（.dex）的Dalvik可执行文件，该格式文件针对小内存使用做了 优化。同时虚拟机是基于寄存器的，所有的类都经由JAVA编译器编译，然后通过SDK中 的 "dx" 工具转化成.dex格式由虚拟机执行。

4. Linux内核(Linux Kernel)

Android的核心系统服务依赖于Linux 2.6 内核，如安全性，内存管理，进程管理， 网络协议栈和驱动模型。 Linux 内核也同时作为硬件和软件栈之间的抽象层。其外还对其做了部分修改，主要涉及两部分修改：

1. Binder (IPC)：提供有效的进程间通信，虽然linux内核本身已经提供了这些功能，但Android系统很多服务都需要用到该功能，为了某种原因其实现了自己的一套。
2. 电源管理：主要是为了省电，毕竟是手持设备嘛，低耗电才是我们的追求。

注:最后附上原博连接懒虫一个V：android系统体系结构，关于谷歌Android源码的目录结构并未一并贴出可在原博查阅

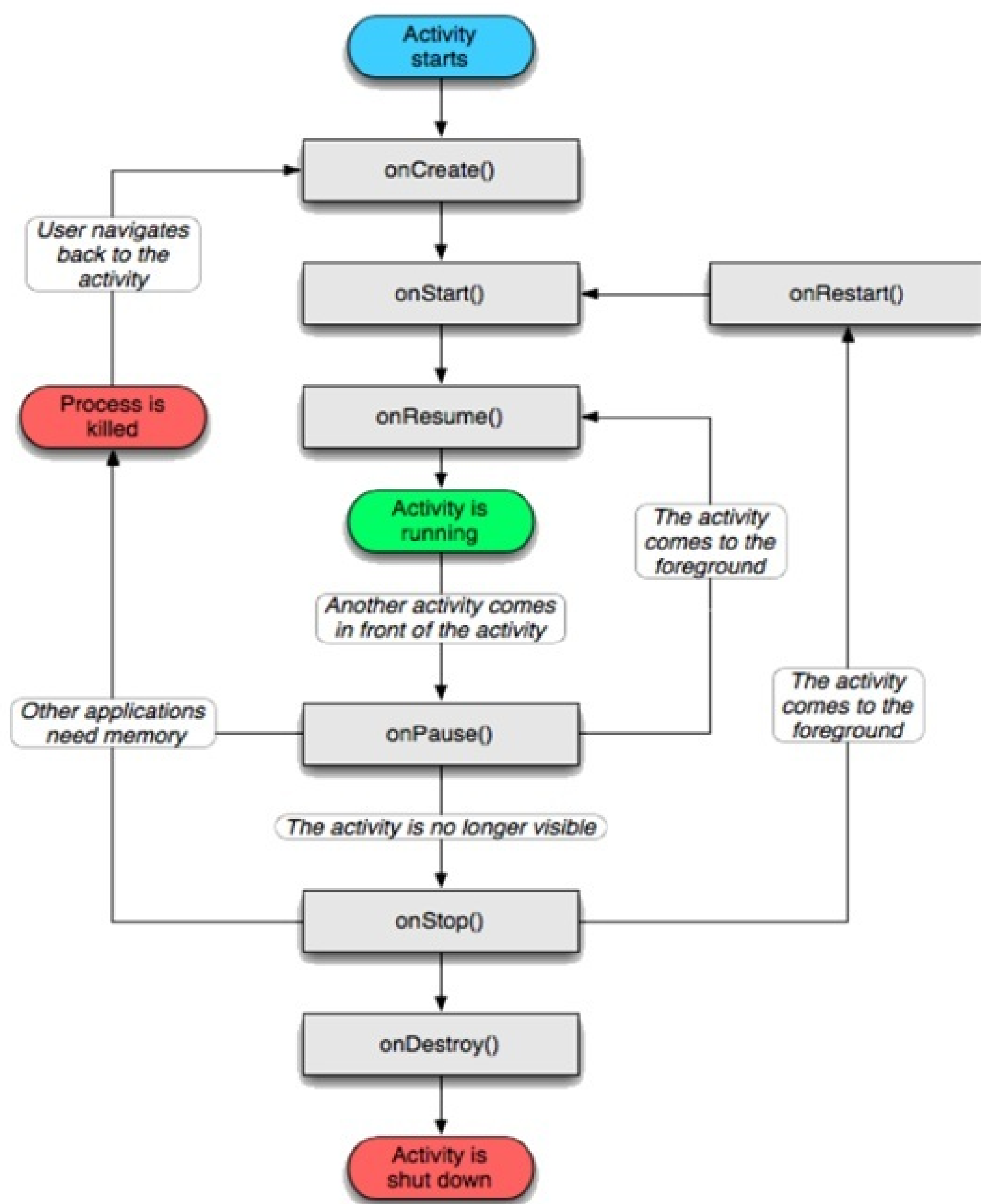
Activity生命周期

总论

了解Activity的生命周期，需要了解：

1. 四种状态
2. 七个重要方法
3. 三个嵌套循环
4. 其他

首先在开头放出生命周期的一张总图：



四种状态

四种状态包括

- 活动（Active/Running）状态
- 暂停(Paused)状态
- 停止(Stopped)状态
- 非活动（Dead）状态

1. 活动（Active/Running）状态

当Activity运行在屏幕前台(处于当前任务活动栈的最上面),此时它获取了焦点能响应用户的操作,属于运行状态，同一个时刻只会有一个Activity 处于活动(Active)或运行(Running)状态。

此状态由onResume()进入，由onPause()退出

2. 暂停(Paused)状态

当Activity失去焦点但仍对用户可见(如在它之上有另一个透明的Activity或Toast、AlertDialog等弹出窗口时)它处于暂停状态。暂停的Activity仍然是存活状态(它保留着所有的状态和成员信息并保持和窗口管理器的连接),但是当系统内存极小时可以被系统杀掉。

此状态由onPause()进入，可能下一步进入onResume()或者onCreate()重新唤醒软件，或者被onStop()杀掉

3. 停止(Stopped)状态

完全被另一个Activity遮挡时处于停止状态,它仍然保留着所有的状态和成员信息。只是对用户不可见,当其他地方需要内存时它往往被系统杀掉。

该状态由onStop()进入，如果被杀掉，可能进入onCreate()或onRestart()，如果彻底死亡，进入onDestroy()

Service生命周期

Service有两种启动方式:

- `startService()` 启动本地服务 `Local Service`
- `bindService()` 启动远程服务 `Remote Service`

远程服务允许暴露接口并让系统内不同程序相互注册调用。Local Service无法抵抗一些系统清理程序如MIUI自带的内存清除。

具体如何防止自己的Service被杀死可以看这个博客Android开发之如何保证Service不被杀掉（broadcast+system/app），已经做到很变态的程度了。此外今天看到如何看待 MIUI 工程师袁军对 QQ 后台机制的评论？，QQ的开启一个像素在前台的做法真的是...呵呵

两种不同的启动方式决定了 `Service` 具有两种生命周期的可能（并非互斥的两种）。概括来说， `Service` 在被创建之后都会进入回调 `onCreate()` 方法，随后根据启动方式分别回调 `onStartCommand()` 方法和 `onBind()` 方法。如果 `Service` 是经由 `bindService()` 启动，则需要所有client全部调用 `unbindService()` 才能将 `Service` 释放等待系统回收。

一张图解释：

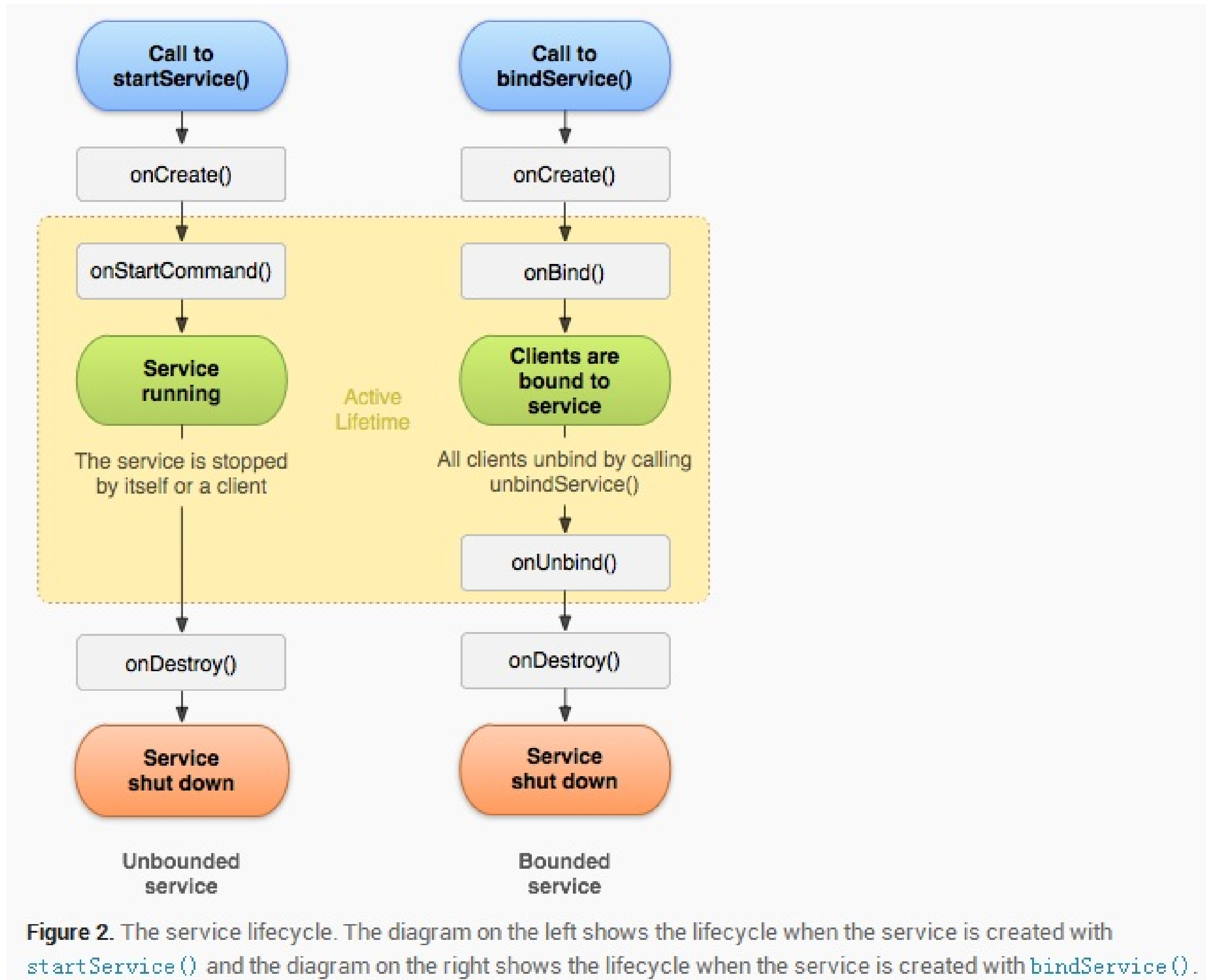


Figure 2. The service lifecycle. The diagram on the left shows the lifecycle when the service is created with `startService()` and the diagram on the right shows the lifecycle when the service is created with `bindService()`.

回调方法的结构下图解释的很明白：

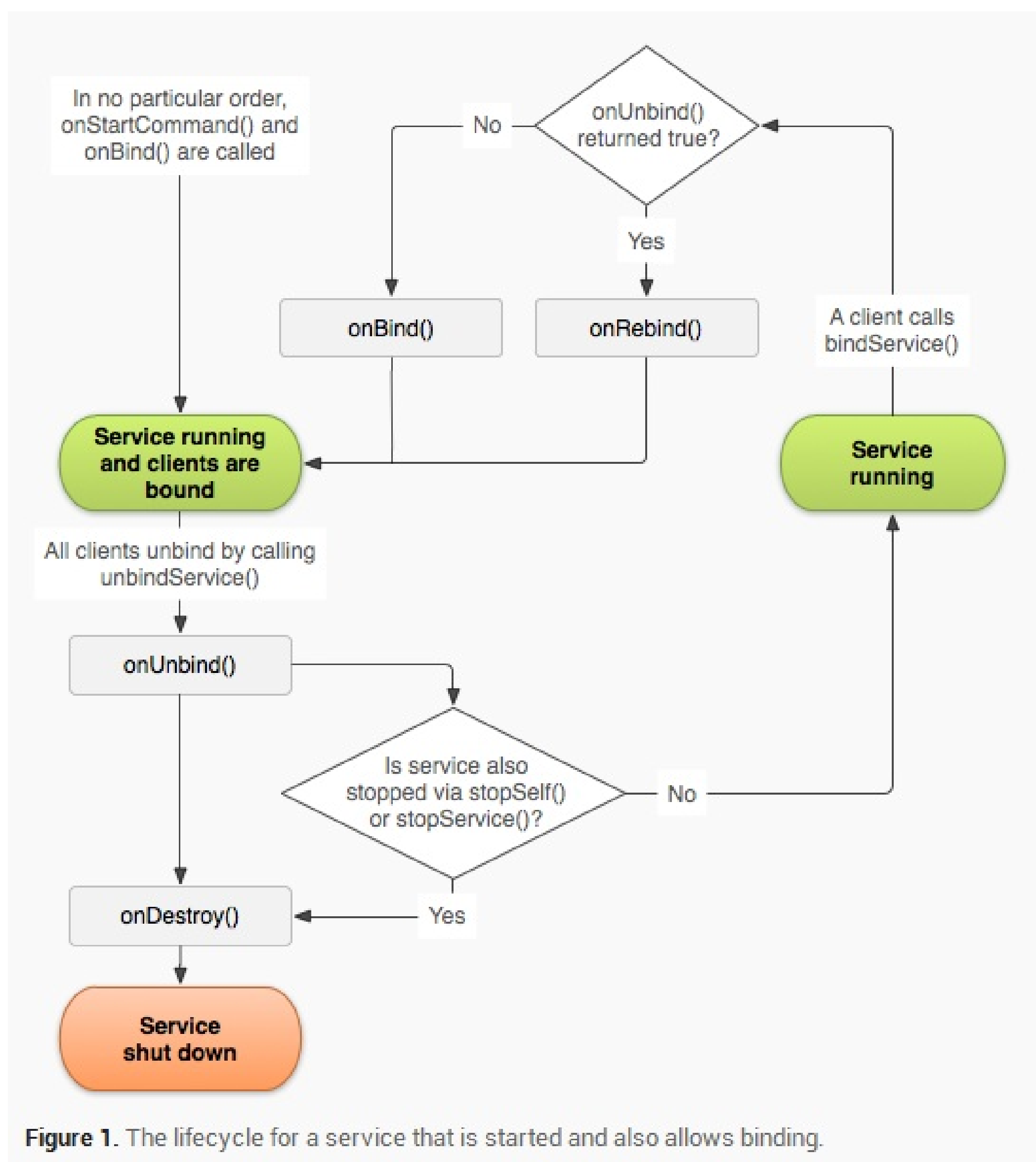


Figure 1. The lifecycle for a service that is started and also allows binding.

参考博客：圣骑士Wind的博客：Android Service的生命周期

Android中的动画

综述

Android中的动画分为补间动画(Tweened Animation)和逐帧动画(Frame-by-Frame Animation)。没有意外的，补间动画是在几个关键的节点对对象进行描述又系统进行填充。而逐帧动画是在固定的时间点以一定速率播放一系列的drawable资源。下面对两种动画进行分别简要说明。

补间动画

补间动画分为如下种

- Alpha 淡入淡出
- Scale 缩放
- Rotate 旋转
- Translate 平移

这些动画是可以同时进行和顺次进行的。需要用到AnimationSet来实现。调用AnimationSet.addAnimation()即可。 实现方法举例:

```
(Button)btn = (Button)findViewById(...);
AnimationSet as = new AnimationSet(false);//新建AnimationSet实例
TranslateAnimation ta = new TranslateAnimation(//新建平移动画实例，在构造函数中传入平移的始末位置
        Animation.RELATIVE_TO_SELF, 0f,
        Animation.RELATIVE_TO_SELF, 0.3f,
        Animation.RELATIVE_TO_SELF, 0f,
        Animation.RELATIVE_TO_SELF, 0.3f);
ta.setStartOffset(0);//AnimationSet被触发后立刻执行
ta.setInterpolator(new AccelerateDecelerateInterpolator());//加入一个加速减速插值器
ta.setFillAfter(true);//动画结束后保持该状态
ta.setDuration(700);//设置动画时长

ScaleAnimation sa = new ScaleAnimation(1f, 0.1f, 1f, 0.1f,//构造一个缩放动画实例，构造函数参数传入百分比和缩放中心
        ScaleAnimation.RELATIVE_TO_SELF, 0.5f,
        ScaleAnimation.RELATIVE_TO_SELF, 0.5f);
sa.setInterpolator(new AccelerateDecelerateInterpolator());//加入一个加速减速插值器
sa.setDuration(700);//设置时长
sa.setFillAfter(true);//动画结束后保持该状态
sa.setStartOffset(650);//AnimationSet触发后650ms启动动画

AlphaAnimation aa = new AlphaAnimation(1f, 0f);//构造一个淡出动画，从100%变为0%
aa.setDuration(700);//设置时长
aa.setStartOffset(650);//AnimationSet触发后650ms启动动画
aa.setFillAfter(true);//动画结束后保持该状态

as.addAnimation(ta);
as.addAnimation(sa);
as.addAnimation(aa);//将动画放入AnimationSet中

btn.setOnClickListener(new OnClickListener(){
  public void onClick(View view){
    btn.startAnimation(as);//触发动画
  }
}
```

该段代码实现了先平移，然后边缩小边淡出。

具体的代码实现需要注意各个参数所代表的含义，比较琐碎，建议阅读文档熟悉。在这里不做过多讲解，文档说的已经很清楚了。
文档连接http://developer.android.com/reference/android/view/animation/Animation.html

逐帧动画

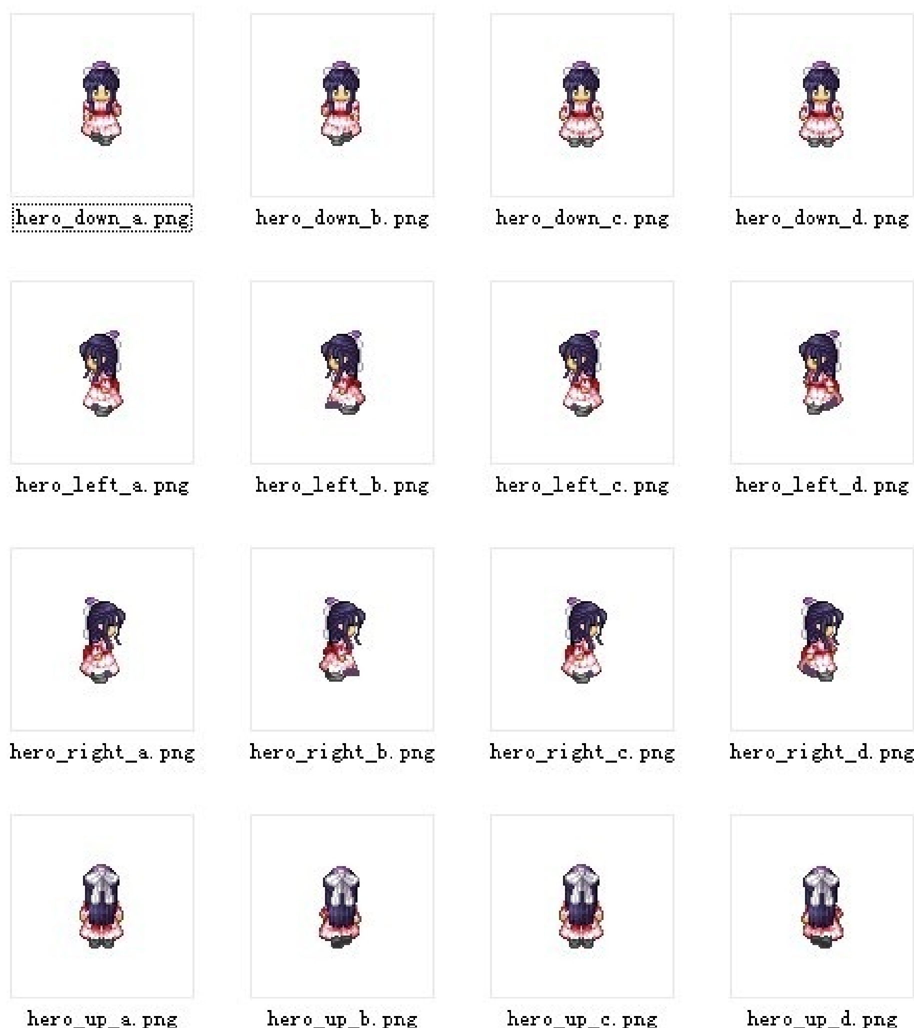
这一部分只涉及非常基础的知识。逐帧动画适用于更高级的动画效果，原因可想而知。我们可以将每帧图片资源放到drawable下然后代码中canvas.drawBitmap(Bitmap, Matrix, Paint)进行动画播放，但这样就将动画资源与代码耦合，如果哪天美工说我要换一下效果就呵呵了。因此我们要做的是将资源等信息放入配置文件然后教会美工怎么改配置文件，这样才有时间去刷知乎而不被打扰^_^。 大致分为两种方法：

- 每一帧是一张png图片中
- 所有动画帧都存在一张png图片中

当然还有的专门的游戏公司有自己的动画编辑器，这里不加说明。

每一帧是一张png

说的就是这个效果：



在animation1.xml文件中进行如下配置：

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot="true"<!-- true表示只播放一次，false表示循环播放 -->
  >
    <item android:drawable="@drawable/hero_down_a" android:duration="70"></item>
    <item android:drawable="@drawable/hero_down_b" android:duration="70"></item>
    <item android:drawable="@drawable/hero_down_c" android:duration="70"></item>
    <item android:drawable="@drawable/hero_down_d" android:duration="70"></item>
</animation-list>
```

在JAVA文件中我们进行如下加载：

```
ImageView animationIV;
AnimationDrawable animationDrawable;

animationIV.setImageResource(R.drawable.animation1);
animationDrawable = (AnimationDrawable) animationIV.getDrawable();
animationDrawable.start();
```

注意动画的播放是按照xml文件中的顺序顺次播放，如果要考虑到循环播放的时候应该写两个xml一个正向一个反向才能很好地循环播放。

所有动画在一张png中

说的就是这个效果：


animation.xml的配置：

```
<key>010001.png</key>
<dict>
    <key>frame</key>
    <string>{{378, 438}, {374, 144}}</string>
    <key>offset</key>
    <string>{-2, 7}</string>
    <key>sourceColorRect</key>
    <string>{{61, 51}, {374, 144}}</string>
    <key>sourceSize</key>
    <string>{500, 260}</string>
</dict>
<key>010002.png</key>
<dict>
    <key>frame</key>
    <string>{{384, 294}, {380, 142}}</string>
    <key>offset</key>
    <string>{1, 7}</string>
    <key>sourceColorRect</key>
    <key>rotate</key>
    <false/>
    <string>{{61, 52}, {380, 142}}</string>
    <key>sourceSize</key>
    <string>{500, 260}</string>
</dict>
…
```

其中：

- frame 指定在原图中截取的框大小；

- offeset 指定原图中心与截图中心偏移的向量；
- rotate若为true顺时针旋转90°；
- sourceColorRect 截取原图透明部分的大小
- sourceSize 原图大小

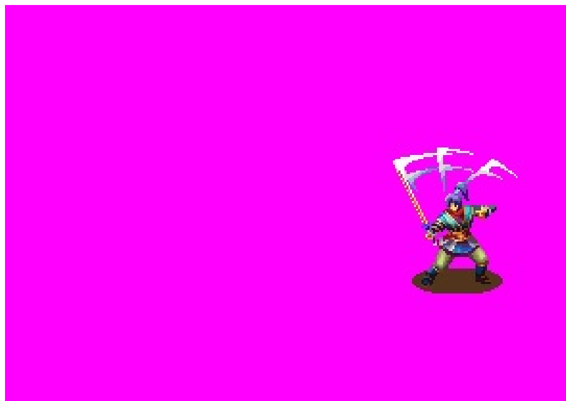
JAVA的加载方式与第一种方法相同。

在使用过程中一定要注意内存资源的回收和drawable的压缩，一不小心可能爆掉。

本文参考博闻：

- .plist中各个key的含义
- Android游戏中的动画制作
- Android研究院值游戏开发
- 用Animation-list实现逐帧动画

最后放一张demo:



Android Activity的Launch Mode

综述

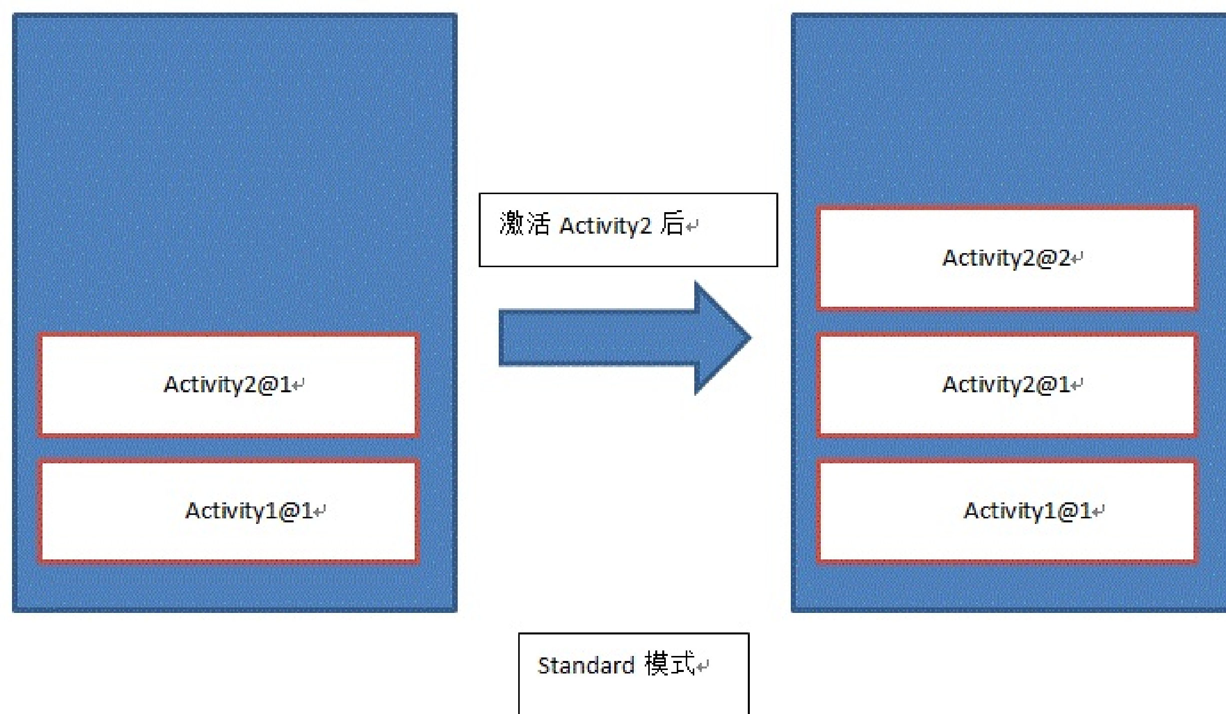
对安卓而言，Activity有四种启动模式，它们是：

- standard 标准模式，每次都新建一个实例对象
- singleTop 如果在任务栈顶发现了相同的实例则重用，否则新建并压入栈顶
- singleTask 如果在任务栈中发现了相同的实例，将其上面的任务终止并移除，重用该实例。否则新建实例并入栈
- singleInstance 允许不同应用，进程线程等共用一个实例，无论从何应用调用该实例都重用

想要感受一下的话写一个小demo，然后自己启动自己再点返回键就看出来了。下面详细说说每一种启动模式

standard

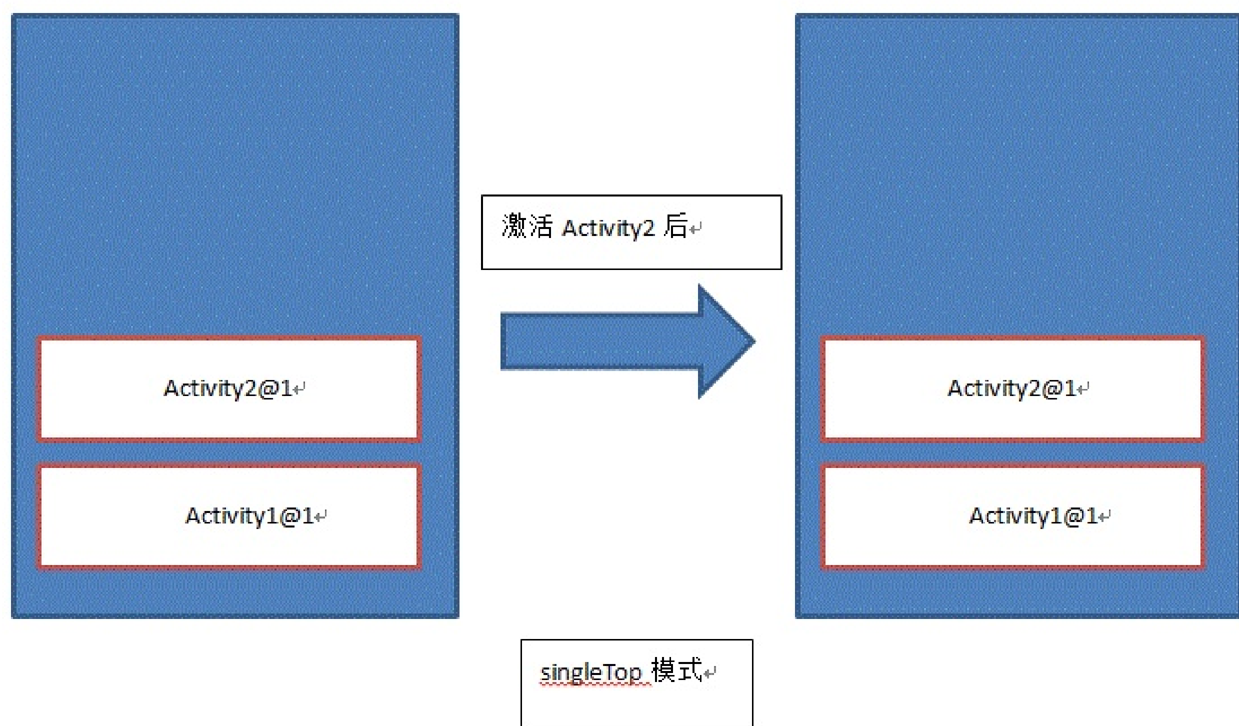
一张图就很好理解



什么配置都不写的话就是这种启动模式。但是每次都新建一个实例的话真是过于浪费，为了优化应该尽量考虑余下三种方式。

singleTop

每次扫描栈顶，如果在任务栈顶发现了相同的实例则重用，否则新建并压入栈顶。

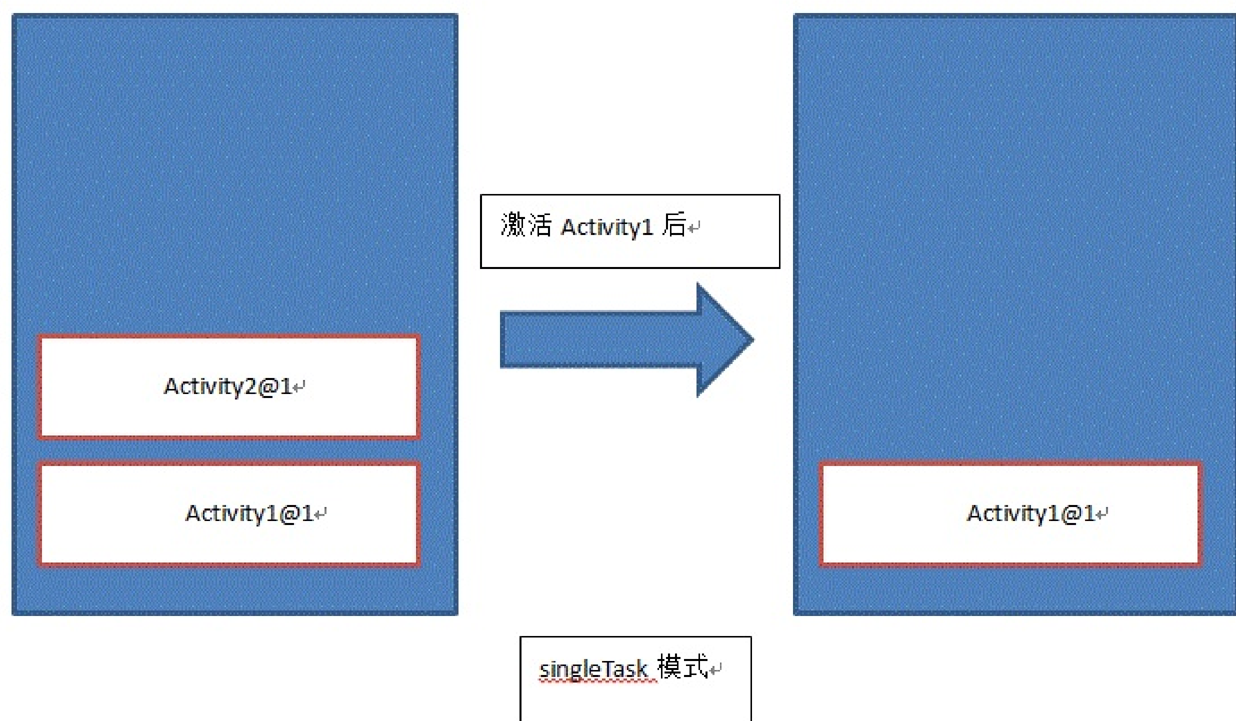


配制方法实在Mainifest.xml中进行：

```
<activity
    android:name=".SingleTopActivity"
    android:label="@string/singletop"
    android:launchMode="singleTop" >
</activity>
```

singleTask

与singleTop的区别是singleTask会扫描整个任务栈并制定策略。上效果图：



使用时需要小心因为会将之前入栈的实例之上的实例全部移除，需要格外小心逻辑。

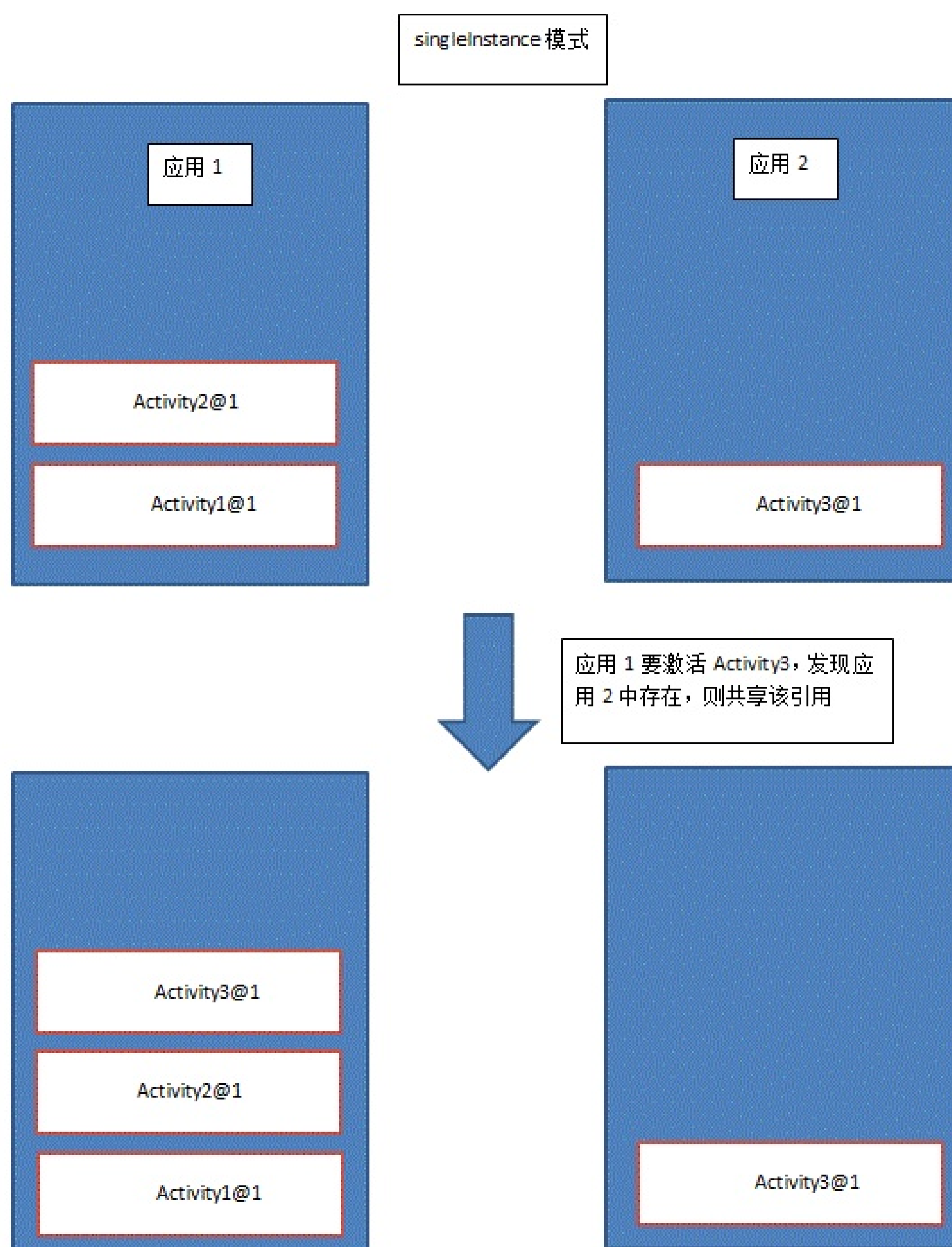
配制方法：

```
<activity
    android:name=".SingleTopActivity"
    android:label="@string/singletop"
    android:launchMode="singleTop" >
</activity>
```

singleInstance

这个的理解可以这么看：在微信里点击“用浏览器打开”一个朋友圈，然后切到QQ再用浏览器开一个网页，再跑到哪里再开一个页面。每次我们都在Activity中试图启动另一个浏览器Activity，但是在浏览器端看来，都是调用了同一个自己。因为使用了singleInstance模式，不同应用调用的Activity实际上是共享的。

上说明图：



配制方法：

```
<activity
    android:name=".SingleTopActivity"
    android:label="@string/singletop"
    android:launchMode="singleTop" >
</activity>
```

参考博客

传送门：

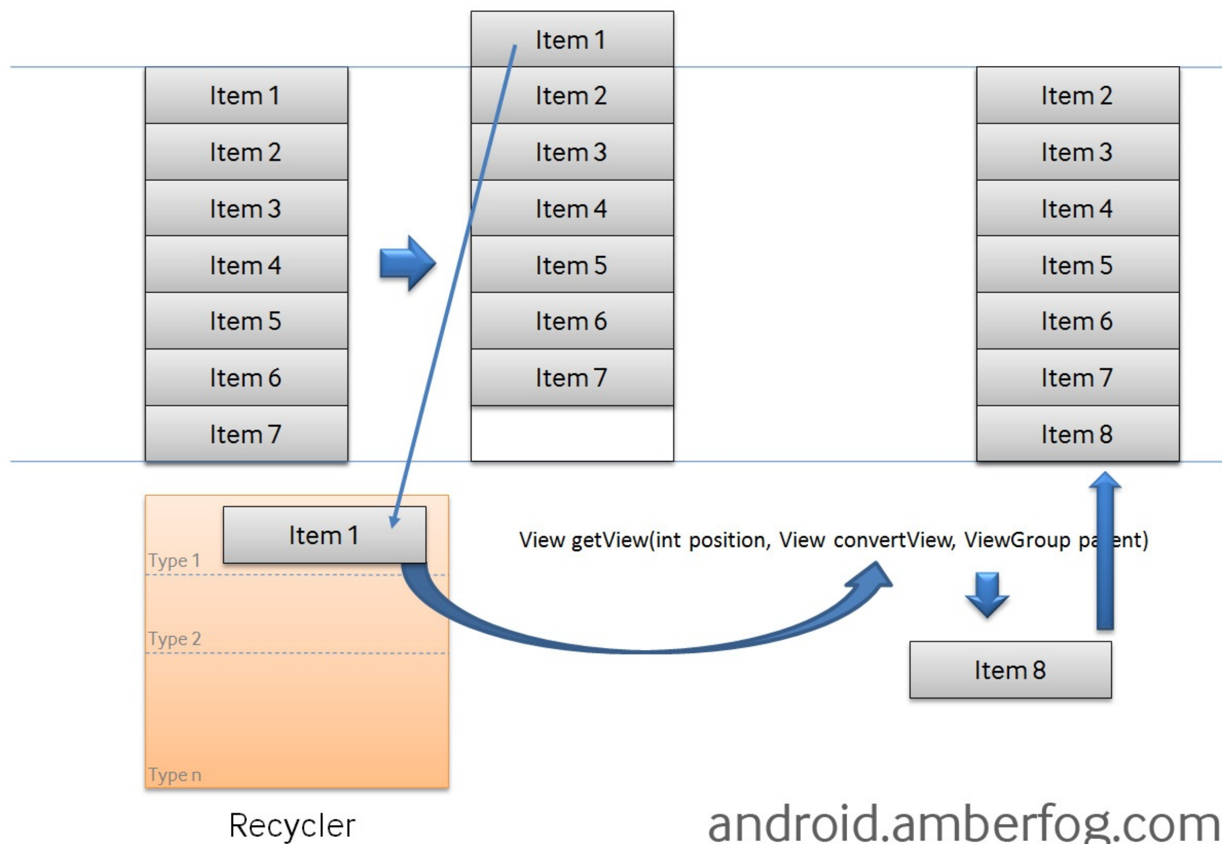
- http://www.cnblogs.com/fanchangfa/archive/2012/08/25/2657012.html
- Android入门：Activity四种启动模式

ListView原理与优化

原理：ListView与Adapter

ListView的实现离不开Adapter。可以这么理解：ListView中给出了数据来的时候，View如何实现的具体方式，相当于MVC中的V；而Adapter提供了相当于MVC中的C，指挥了ListView的数据加载等行为。

提一个问题：假设ListView中有10W个条项，那内存中会缓存10W个吗？答案当然是否定的。那么是如何实现的呢？下面这张图可以清晰地解释其中的原理:



可以看到当一个View移出可视区域的时候，设为View1，它会被标记Recycle，然后可能：

- 新进入的View2与View1类型相同，那么在getView方法传入的convertView就不是null而就是View1。换句话说，View1被重用了
- 新进入的View2与View1类型不同，那么getView传入的convertView就是null，这是需要new一个View。当内存紧张时，View1就会被GC

ListView的优化(以异步加载Bitmap优化为例)

首先概括的说ListView优化分为三级缓存:

- 内存缓存

- 文件缓存
- 网络读取

简要概括就是在getView中，如果加载过一个图片，放入Map类型的一个MemoryCache中(示例代码使用的是Collections.synchronizedMap(new LinkedHashMap(10, 1.5f, true))来维护一个试用LRU的堆)。如果这里获取不到，根据View被Recycle之前放入的TAG中记录的uri从文件系统中读取文件缓存。如果本地都找不到，再去网络中异步加载。

这里有几个注意的优化点：

1. 从文件系统中加载图片也没有内存中加载那么快，甚至可能内存中加载也不够快。因此在ListView中应设立busy标志位，当ListView滚动时busy设为true，停止各个view的图片加载。否则可能会让UI不够流畅用户体验度降低。
2. 文件加载图片放在子线程实现，否则快速滑动屏幕会卡
3. 开启网络访问等耗时操作需要开启新线程，应使用线程池避免资源浪费，最起码也要用AsyncTask。
4. Bitmap从网络下载下来最好先放到文件系统中缓存。这样一是方便下一次加载根据本地uri直接找到，二是如果Bitmap过大，从本地缓存可以方便的使用Option.inSampleSize配合Bitmap.decodeFile(ui, options)或Bitmap.createScaledBitmap来进行内存压缩

**原博文有非常好的代码示例: Listview异步加载图片之优化篇（有图有码有解释）非常值得看看。

此外Github上也有仓库：https://github.com/geniusgithub/SyncLoaderBitmapDemo

Android中的Thread, Looper和Handler机制(附带HandlerThread与AsyncTask)

Thread，Looper和Handler的关系

与Windows系统一样，Android也是消息驱动型的系统。引用一下消息驱动机制的四要素：

- 接收消息的“消息队列”
- 阻塞式地从消息队列中接收消息并进行处理的“线程”
- 可发送的“消息的格式”
- “消息发送函数”

与之对应，Android中的实现对应了

- 接收消息的“消息队列” ——【MessageQueue】
- 阻塞式地从消息队列中接收消息并进行处理的“线程” ——【Thread+Looper】
- 可发送的“消息的格式” ——【Message】
- “消息发送函数”——【Handler的post和sendMessage】

一个 `Looper` 类似一个消息泵。它本身是一个死循环，不断地从 `MessageQueue` 中提取 `Message` 或者Runnable。而 `Handler` 可以看做是一个 `Looper` 的暴露接口，向外部暴露一些事件，并暴露 `sendMessage()` 和 `post()` 函数。

在安卓中，除了 `UI线程` / `主线程` 以外，普通的线程(先不提 `HandlerThread`)是不自带 `Looper` 的。想要通过UI线程与子线程通信需要在子线程内自己实现一个 `Looper` 。开启Looper分三步走：

1. 判定是否已有 `Looper` 并 `Looper.prepare()`
2. 做一些准备工作(如暴露handler等)
3. 调用 `Looper.loop()` ，线程进入阻塞态

由于每一个线程内最多只可以有一个 `Looper` ，所以一定要在 `Looper.prepare()` 之前做好判定，否则会抛出 `java.lang.RuntimeException: Only one Looper may be created per thread` 。为了获取Looper的信息可以使用两个方法：

- Looper.myLooper()
- Looper.getMainLooper()

`Looper.myLooper()` 获取当前线程绑定的Looper，如果没有返回 `null` 。 `Looper.getMainLooper()` 返回主线程的 `Looper` ,这样就可以方便的与主线程通信。注意：在 **`Thread`** 的构造函数中调用 **`Looper.myLooper`** 只会得到主线程的 **`Looper`** ，因为此时新线程还未构造好

下面给一段代码，通过Thread，Looper和Handler实现线程通信：

MainActivity.java

```
public class MainActivity extends Activity {
    public static final String TAG = "Main Acticity";
    Button btn = null;
    Button btn2 = null;
    Handler handler = null;
    MyHandlerThread mHandlerThread = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        btn = (Button)findViewById(R.id.button);
        btn2 = (Button)findViewById(R.id.button2);
        Log.d("MainActivity.myLooper()", Looper.myLooper().toString());
        Log.d("MainActivity.MainLooper", Looper.getMainLooper().toString());


        btn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                mHandlerThread = new MyHandlerThread("onStartHandlerThread");
                Log.d(TAG, "创建myHandlerThread对象");
                mHandlerThread.start();
                Log.d(TAG, "start一个Thread");
            }
        });

        btn2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                if(mHandlerThread.mHandler != null){
                    Message msg = new Message();
                    msg.what = 1;
                    mHandlerThread.mHandler.sendMessage(msg);
                }

            }
        });
    }
}
```

MyHandlerThread.java

```
public class MyHandlerThread extends Thread {
    public static final String TAG = "MyHT";

    public Handler mHandler = null;

    @Override
    public void run() {
        Log.d(TAG, "进入Thread的run");
        Looper.prepare();
        Looper.prepare();
        mHandler = new Handler(Looper.myLooper()){
            @Override
            public void handleMessage(Message msg){
                Log.d(TAG, "获得了message");
                super.handleMessage(msg);
            }
        };
        Looper.loop();
    }
}
```

HandlerThread 和 AsyncTask

HandlerThread

Android为了方便对 `Thread` 和 `Handler` 进行封装，也就是 `HandlerThread` 。文档中对 `HandlerThread` 的定义是：

> Handy class for starting a new thread that has a looper. The looper can then be used to create handler classes. Note that start() must still be called.

`HandlerThread` 继承自 `Thread` ，说白了就是 `Thread` 加上一个一个 `Looper` 。分析下面的代码:

```
public class MyHandlerThread extends HandlerThread{
    @Override
    public void run(){
        if(Looper.myLooper == null){
            Looper.prepare();
        }
        super.run();
    }
}
```

会抛出 `java.lang.RuntimeException: Only one Looper may be created per thread` 错误。如果我们把super.run()注释掉就不会有这样的错误。显然在 `super.run()` 中进行了Looper的绑定。

AsyncTask

AsyncTask是谷歌对Thread和Handler的进一步封装，完全隐藏起了这两个概念，而用 `doInBackground(Params... params)` 取而代之。但需要注意的是AsyncTask的效率不是很高而且资源代价也比较重，只有当进行一些小型操作时为了方便起见使用。这一点在官方文档

写的很清楚:

> AsyncTask is designed to be a helper class around Thread and Handler and does not constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most.) If you need to keep threads running for long periods of time, it is highly recommended you use the various APIs provided by the java.util.concurrent package such as Executor, ThreadPoolExecutor and FutureTask.

由于使用比较简单应该不需要细说。如有需要会在未来更新。

1. 什么是ANR，如何避免
2. [[ListView原理与优化|ListView-Optimize]]
3. ContentProvider实现原理
4. 介绍Binder机制
5. 匿名共享内存，使用场景
6. 如何自定义View，如果要实现一个转盘圆形的View，需要重写View中的哪些方法？(onLayout,onMeasure,onDraw)
7. Android事件分发机制
8. Socket和LocalSocket
9. [[如何加载大图片|Android-Large-Image]]
10. HttpClient和URLConnection的区别，怎么使用https
11. Parcelable和Serializable区别
12. Android里跨进程传递数据的几种方案。(Binder,文件[面试官说这个不算],Socket,匿名共享内存（Anonymous Shared Memory))
13. 布局文件中，layout_gravity 和 gravity 以及 weight的作用。
14. ListView里的ViewType机制
15. TextView怎么改变局部颜色(SpannableString或者HTML)
16. Activity A 跳转到 Activity B，生命周期的执行过程是啥？(此处有坑 ActivityA的OnPause和ActivityB的onResume谁先执行)
17. Android中Handler声明非静态对象会发出警告，为什么，非得是静态的？(Memory Leak)
18. ListView使用过程中是否可以调用addView(不能，话说这题考来干啥。。。)
19. [[Android中的Thread, Looper和Handler机制(附带HandlerThread与AsyncTask)|Android-handler-thread-looper]]
20. Application类的作用
21. View的绘制过程
22. 广播注册后不解除注册会有什么问题？(内存泄露)
23. 属性动画(Property Animation)和补间动画(Tween Animation)的区别，为什么在3.0之后引入属性动画(官方解释：调用简单)
24. 有没有使用过EventBus或者Otto框架，主要用来解决什么问题，内部原理
25. 设计一个网络请求框架(可以参考Volley框架)
26. 网络图片加载框架(可以参考BitmapFun)
27. Android里的LRU算法原理
28. BrocastReceive里面可不可以执行耗时操作?
29. Service onBindService 和startService 启动的区别